TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

CSR-24-01

Hardware-Supported Test Environment Analysis for CAN Message Communication

Seyhmus Akaslan · Ariane Heller · Wolfram Hardt

Juni 2024

Chemnitzer Informatik-Berichte

Hardware-Supported Test Environment Analysis for CAN Message Communication

Master Thesis

Submitted in Fulfillment of the
Requirements for the Academic Degree
M.Sc.

Dept. of Computer Science
Chair of Computer Engineering

Submitted by: Seyhmus AKASLAN
Student ID: 725512
Date: 2024-04-25

Supervising tutors: Prof. Dr. W. Hardt
Dr. Ariane Heller
Jörg Höninger
Evangelos Tsigos

Abstract

Recent innovations in technology and demands for more functionality increased software size in cars up to 100 million lines of code. Explosion in software size is accompanied by an increased number of ECUs. Testing of software became more complex than ever. To be able to test the exact timing behavior of a software, it needs to be put on actual hardware. HIL test benches have become an indispensable part of ECU testing. The hard part of ECU testing is their dependency on each other. ECUs communicate to each other by passing information in the form of CAN messages. This makes testing a single ECU alone without its environment impossible. Because of their dependencies on each other they need to be integrated first to be tested. Before HIL benches those tests were done either on vehicles or in integration labs where ECUs are connected to each other in a lab environment. Advances in software science brought up an invention known as rest-bus simulation. In some sources it is also known as residual bus or rest of the bus simulation. HIL platforms simulate missing nodes and messages as if the actual hardware is there. HIL platforms blend the real world and simulated ones in real time. Device under test thinks it is present in a vehicle. Established HIL platforms solved many problems that existed in software projects. However, because of their cost, only a small number of such platforms are affordable within a company. Developers need to wait in long queues to test their ECUs. These waiting times can be even longer in agile software development methodologies due to their frequent testing needs. It is believed that front-loading tests before the HIL is the solution to this problem. The aim of this thesis is to investigate alternative small form factor HIL platforms which could be placed on every developer's desk. To place the proposed solution at every desk, the solution must be affordable and portable. The solution will hopefully reduce queues accumulating behind established HIL platforms and shorten testing times. For this end, state-of-the-art HIL solutions will be investigated. Afterwards, a working proof-of-concept will be demonstrated in the form of residual bus simulation and gateway application. Portability of the solution is a must for the gateway application. Test engineers use gateway applications to alter some signal values either on HIL or on-vehicle. Only small and easy to carry solutions are feasible for on-vehicle testing. It will be shown that the proposed solution will reduce the testing time and testing cost. In addition to them, an increase in parallelism, testing frequency, and software quality will be observed by bringing testing equipment to every developer's desk.

Keywords: HIL, Hardware-in-the-loop, CAN, CAN FD, CAPL, CANoe, ECU, Fuel Cell, Controller, Simulation, Analysis, Gateway, Test, Automation

Acknowledgements

I would like to use this opportunity to express my gratitude to everyone who helped me to prepare this thesis. Starting from the university, I am grateful to my internal supervisors Prof. Wolfram Hardt, Ariane Heller, and Mohamed Salim Harras for their time and effort. I am also thankful to my external supervisors at the Cellcentric company, especially Jörg Höninger and Evangelos Tsigos. It would be impossible to finish this thesis in time without them. Finally, I would like to thank all my colleagues at Cellcentric for their support during the whole time I spent over there. They helped me by answering questions and setting up hardware in the office. The list is so long that I cannot list all the names here. Overall, I enjoyed doing my thesis in a company environment and surely, I benefited a lot from them.

Table of Contents

Abstract 1
Acknowledgements 2
Table of Contents 3
List of Figures 4
List of Tables 6
List of Abbreviations 7
1 Introduction 8
2 Technical Background 14
2.1 Software Testing 14
2.2 Test Automation 20
2.3 Simulation and Testing Tools 23
2.4 CAN Communication 27
2.5 Diagnostics 35
3 State of the Art 39
3.1 RTOS Based HIL Platform 41
3.2 COTS Based HIL Platform 43
3.3 COTS and RTA Based HIL Platform 44
4 Methodology 47
4.1 Latency 48
4.2 Periodicity 54
4.3 Rest-bus Simulation 61
4.4 Gateway Application 77
5 Results and Evaluation 81
6 Conclusion 85
7 Bibliography 87

List of Figures

Figure 1 Evolution of E/E architecture [2]. 9
Figure 2 Relationship among MIL, SIL, HIL etc. [11]. 11
Figure 3 V-model of functional safety for software development process [15] [19]....15
Figure 4 Software testing levels [5]. 16
Figure 5 Relationship between testing time and percentage of undiscovered defects [23]. 20
Figure 6 Test module execution. 22
Figure 7 A test report is generated after module execution. 23
Figure 8 Trace window of CANoe [28]. 24
Figure 9 CANoe Graphics window showcasing 20ms periodic message on CAPL-on-board. 25
Figure 10 CAPL Browser screen [30]. 26
Figure 11 Common development stages in CANoe [28]. 26
Figure 12 Interactive Generator (IG) window in CANoe [28]. 27
Figure 13 CAN Standard frame format [32]. 29
Figure 14 CAN Extended frame format [21]. 29
Figure 15 CAN frame CRC field [21]. 30
Figure 16 CAN Error frame format [32]. 30
Figure 17 Bit stuffing for standard frame format [5]. 31
Figure 18 CAN FD base frame format [34]. 33
Figure 19 CAN FD extended frame format [34]. 33
Figure 20 CAPL Browser with diagnostic event handlers [35]. 35
Figure 21 Fault memory window in CANoe [35]. 36
Figure 22 Signal Level HIL simulator [17]. 39
Figure 23 Signal flow in HIL simulation [16]. 42
Figure 24 Cost effective HIL platform based on COTS software and hardware. 43
Figure 25 Cost effective HIL platform based on COTS and RTA. 45
Figure 26 Conceptual representation of the Latency Experiment. 49
Figure 27 Latency vs Time graph of messages, cycle time is 10ms. 51
Figure 28 Latency vs Time graph of messages, cycle time is 10ms. 51
Figure 29 Artificial load is created by a generator. 53
Figure 30 Simulation Setup of periodicity experiment. 54
Figure 31 Periodicity vs Time graph of messages, cycle time is 10ms. 56
Figure 32 Periodicity graph of 20ms messages under no load. 56
Figure 33 Periodicity graph of 20ms messages under load. 57

Figure 34 Non RTOS HIL shows period fluctuations for a message cycle time is 20ms [38]. 59
Figure 35 CAPL-on-board configuration of a node. 60
Figure 36 Periodicity measurement under no load in CAPL-on-board mode cycle time 20ms. 60
Figure 37 RTA (Real-time adapter) HIL for 20ms cycle time messages [38]. 61
Figure 38 Layout of Rest-bus simulation. 63
Figure 39 CAN1 Communication Matrix in DBC file format. 64
Figure 40 Simulation layout contains simulated nodes. 64
Figure 41 Some captured signals in rest-bus simulation. 66
Figure 42 Statistics window of CAN interfaces. 67
Figure 43 Rest-bus simulation CAN1 Data window. 68
Figure 44 Panels created to ease message manipulation by hand. 69
Figure 45 Panel allows easily enable and disable simulated nodes. 70
Figure 46 DTCs generated in the default state of the simulation. 71
Figure 47 Test case grouping and setup window. 73
Figure 48 A test module and its test cases are executed. 75
Figure 49 The test report is optionally generated after test execution finalize. 76
Figure 50 Overall layout of Gateway simulation. 77
Figure 51 Panel allows easy management of sensor signals. 78
Figure 52 Data view of incoming sensory messages. 79
Figure 53 Data view of outgoing altered messages. 80

List of Tables

Table 1 Classic CAN extended frame stuff bits calculation. 31
Table 2 CAN FD stuff bit calculation for extended frame.....34
Table 3 CAPL test script checking a DTC. 37
Table 4 CAPL test script for DTC 0x1F0028. 38
Table 5 Comparison table of state-of-the-art HIL platforms.....45
Table 6 CAPL script Node A in latency test.....50
Table 7 CAPL script of Node B in latency test.....50
Table 8 Latency tests, bus is empty, cycle time is 10ms. 51
Table 9 Latency tests for different cycle times, bus is empty.....52
Table 10 Latency tests for different cycle times under artificial load.....53
Table 11 Latency test for 10ms periodic message under artificial load. 53
Table 12 CAPL script of Node 1 in periodicity tests.....54
Table 13 CAPL script of Node 2 in periodicity tests.....55
Table 14 Period vs Time graph for cyclic messages under no load. 57
Table 15 Periodicity vs Time under load. 58
Table 16 CAPL-on-Board Periodicity vs Time measurements under no load.....61
Table 17 CAPL implementation of some timers and messages. 64
Table 18 CAPL implementation of some event handlers.....65
Table 19 Statistical values of some captured signals in rest-bus simulation. 67
Table 20 Disabled nodes causes DTCs. 70
Table 21 DTC Status Byte [36].....71
Table 22 A sample test case showcasing existence and period check. 74

List of Abbreviations

ECU	Electronic Control Unit
FCU	Fuel Cell ECU
RTA	Real Time Adapter
HIL	Hardware-in-the-loop
SIL	Software-in-the-loop
MIL	Model-in-the-loop
PIL	Processor-in-the-loop
XIL	X-in-the-loop
DTC	Diagnostic Trouble Code
DCM	Diagnostic Communication Manager
UDS	Unified Diagnostic Services
CAN	Controller Area Network
RCP	Rapid Control Prototyping
ACC	Adaptive Cruise Control
ABS	Anti Lock Braking System
AUTOSAR	Automotive Open System Architecture
HPC	High performance ECU/Computer
E/E Architecture	Electrical and Electronic Architecture
E2E	End to end
SOA	Service Oriented Architecture
ISO	International Organization for Standardization
COTS	Commercial off-the-shelf
TP	Transport Protocol
NvM	NVRAM Manager
DEM	Diagnostic Event Manager
SUT	System under Test
DUT	Device under Test
CAPL	CAN Access Programming Language
API	Application Programming Language
OEM	Original Equipment Manufacturer
CAN FD	CAN Flexible Data-Rate

1 Introduction

Rapidly growing demand for software such as advanced driver assist systems increased not only the number of lines in the software but also the number of ECUs. Estimation for the number of lines of all software is already way past 100 million, while the number of ECUs reached over 70 [1] [2] [3] [4] [5] [6]. Traditionally each new functionality had to be placed in its own ECU. In the early days the number of ECUs was small, and their dependencies were limited. In today's cars, however, those functionalities are distributed over several ECUs. This results in increased dependency on each other. X-by-wire and ACC (Adaptive Cruise Control) systems, for example, need several ECUs to work closely to function. ECUs use communication lines such as CAN bus to pass messages to each other. The increased number of ECUs and their dependency poses lots of problems while testing, updating, and adding a new feature. Testing of such a system is another matter engineers need to deal with. Fortunately, industry has a solution to all those problems. Some of these solutions are simplification of automotive E/E architecture, using consolidation, containerization, common application development frameworks, high speed communication networks, service-oriented architectures and finally HIL testing.

Classic E/E architecture is divided into multiple domains based on real-time capability, computation power and data throughput needs. There are vehicle-centric functional domains such as power train, chassis, safety systems and passenger centric domains such as multimedia, body, HMI (human machine interface) [5]. Modern architectures such as zonal E/E architectures and consolidation are new solutions aimed to reduce the number of ECUs and simplify vehicle networks. Zonal architecture decouples functionality from underlying hardware. A few high-performance ECUs (HPCs) build the central backbone of the architecture while real-time capable ECUs are put closer to the zones where sensors and actuators are located. Figure 1 represents evolutionary progress of automotive E/E architecture. Simplified architecture also reduces cabling as well as hardware. HPCs consolidate many smaller ECUs into themselves by using virtualization technologies. Multiple ECUs can reside in one single HPC enclosed in containers. Containers allow ECUs to freely move around in the vehicle architecture and hence decouple them from hardware. Thanks to this architecture software updates turn into individual container updates. New functionality can be added to the vehicle either by adding a new container to the system or by updating an existing container. Traditional solution was to throw a new ECU to add a new functionality. Adding a new ECU to an existing car after development is finished was nearly impossible. That would cause a lot of software modification in the

communication matrix and interfaces. The new service-oriented architecture solves this problem by using service discovery protocol SOME/IP (Scalable Service-Oriented Middleware over IP) over Ethernet. The freedom to easily add new functionality to an existing car by only adding new software is a new paradigm known as software defined vehicle in literature. Moreover, new functionality can be transferred to the cars without bringing them to the service points by using over-the-air (OTA) updates. Overall zonal architecture is preferred over classic distributed architecture when more computation power, high transmission rates and ability to add future software easily to existing architecture is desired [1] [4] [7] [6].

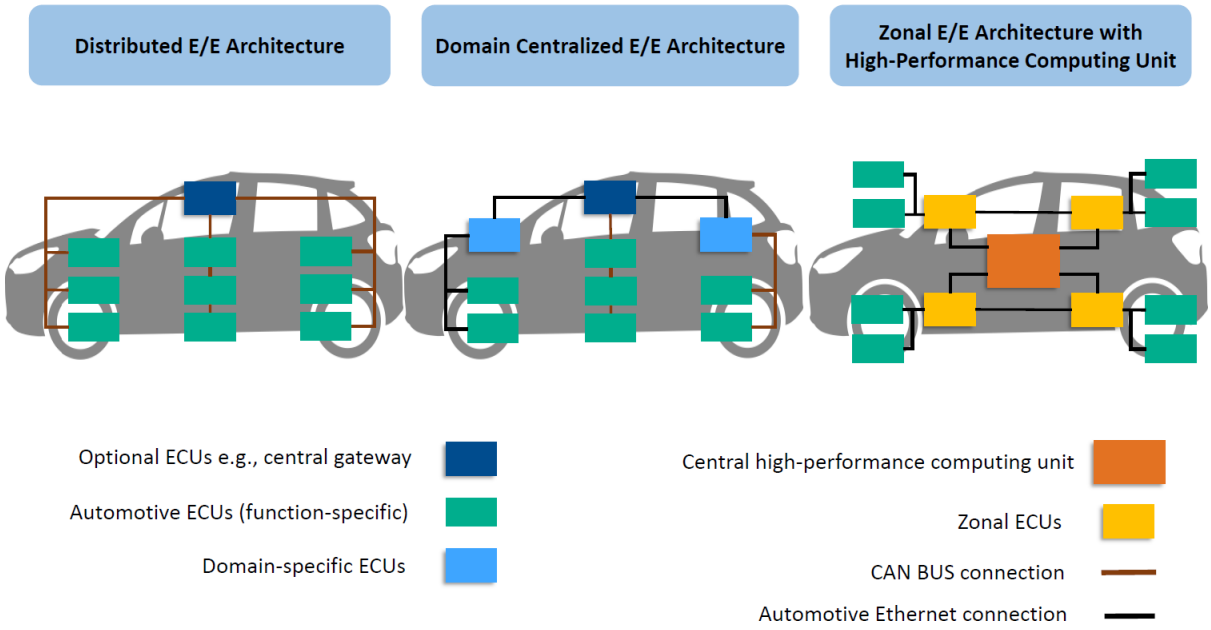

Figure 1 Evolution of E/E architecture [2].

AUTOSAR is a vehicle software framework in use for over 20 years trying to manage complexity of software running on ECUs. Complexity arises due to the growing number of software, diversity of networks and hardware. AUTOSAR comes in two flavors. The classic platform is for embedded hard real-time systems and supports safety integrity level up to ASIL-D whereas the adaptive platform is for soft real-time dynamic systems, based on POSIX and requires high computing power. The safety integrity level supported by the adaptive platform is up to ASIL-B. AUTOSAR defines layered architecture. The lowest layer is MCAL (Microcontroller abstraction layer) which provides abstraction to devices and can be regarded as device drivers. Microcontroller, communication, memory, and IO drivers belong to MCAL. ECU abstraction layer is the middle layer and provides services for device drivers to the top layer. The top layer is the service layer containing services such as NvM (NVRAM Manager), AUTOSAR OS, DCM (Diagnostic Communication Manager) and DEM (Diagnostic Event Manager). Each module has a well defined interface. The service layer still needs direct access to the hardware for features such as time-sliced scheduling etc. Besides three software

layers, one additional layer called complex device drivers needs direct hardware access is introduced to extend architecture to cover non-standardized use cases. AUTOSAR with its layered architecture, provides hardware independence, promotes collaboration among vendors, software reuse, software portability, and interoperability of software components with well-defined open interfaces. Development methodology comes with AUTOSAR encourages model-driven development of software, hardware, and network topology. They can be modeled in metamodel language ARXML where rules are defined by OMG (Object Management Group) [8] [4] [5].

SAE (Society for Automotive Engineers) classifies communication protocols based on their speeds. Class A networks have bandwidth lower than 10Kbps. This class of network is usually used in the body domain (windows, lighting, door locks, etc.). Class B has bandwidth in the range of 10Kbps to 125Kbps. Low speed CAN correspond to this class. Class C networks operate in the range 125Kbps to 1Mbps, and class D networks operate speeds above 1Mbps. Class C networks correspond to high-speed CAN and are used in the powertrain and chassis domain where speed and real-time requirements are higher. Class D networks on the other hand are devoted to multimedia, safety critical applications and gateway systems. We are living in the era of connected cars. The current capacity of in-car networks can't handle the high volume of data exchange between cars and the infrastructure due to speed limitations and broadcast nature of the CAN bus. Automotive ethernet is proposed to solve ever increasing high bandwidth demands and transforming vehicle architecture from statically defined signal-oriented CAN based communication to dynamically defined service-oriented architecture (SOA). SOA architecture states that data is delivered to a node only if the node registers itself to the service. Connecting the vehicle to the cloud implicates security concerns. Cybersecurity is another aspect that shouldn't be taken lightly in the era of connected cars. A successful attack on the safety critical functions of a vehicle might pose danger to the passenger as well as vehicle. Firewalls, IDS (Intrusion detection system) and access control mechanisms are developed to mitigate security concerns [4] [5].

Depending on its place on the V-model, testing techniques get different names. MIL (Model-in-the-loop) is the earliest possible testing stage. Entire system, both plant model and controller models are tested in a simulated environment [9]. SIL (Software-in-the-loop) executes software components on a host computer. This test is preferred when actual hardware is not available. PIL (Processor-in-the-loop) runs software on a development board where the processor is the same as the actual hardware. Development boards may have missing peripherals and those missing need to be

simulated by software-stubs. HIL (Hardware-in-the-loop) uses actual hardware and interfaces. The plant model is simulated on the HIL and interacts with actual hardware through interfaces. Controller can't differentiate the simulation from the actual plant [10]. VIL (Vehicle-in-the-loop) tests are mostly suited to ADAS (advanced driver assist systems) and AD (Autonomous Driving) systems. These controllers need to access environmental resources. The environment is moved to a virtual platform in VIL. In the virtual environment sensors, actuators, and test scenarios can be simulated. In other words, VIL is a fusion of real vehicles and virtual environments. VIL reduces training time of neural networks to considerable degrees [11].

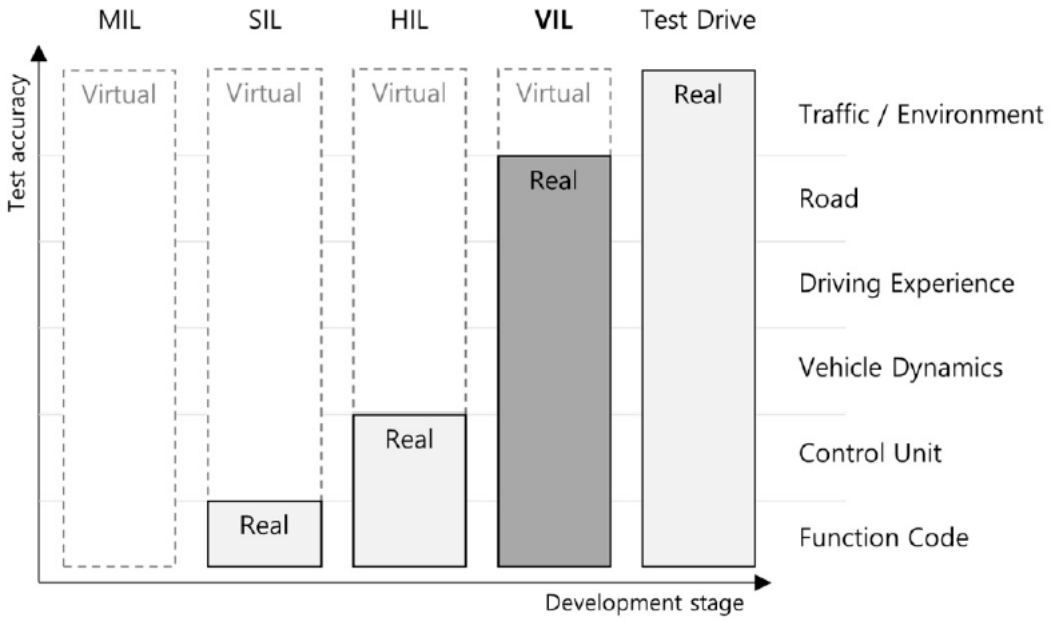

Figure 2 Relationship among MIL, SIL, HIL etc. [11].

Although the size of software keeps increasing, testing of ECUs is still limited to HIL and on-vehicle environments. Faults that require a redesign or an interface change must be dealt with in the early stages of development, preferably before the HIL. Frontloading tests to SIL and MIL stages has a lot of benefits. The most obvious one is increased parallelism. Every developer can run their tests on the desktop without interrupting others. Another advantage is the virtualization as such testing platforms can be moved to the cloud. Cloud provides unlimited resources and faster simulation speeds, even faster than real-time. Software testing needs to be done as early as possible to reduce the cost of fixing faults in later stages. Frontloading is a trend that aims to reduce testing time by moving tests from upper levels to the more controlled environments like HIL or SIL stages. Testing at lower levels of V-model can provide increased levels of parallelization, testing frequency and at the same time reduces costs of fixing mistakes while they are still cheaper to fix. Testing done at higher levels of V-model has increased real-time behavior but no parallelization. Real prototypes are

expensive to manufacture. A fault in the tested software may damage the system bench or prototype. Engine and ACC controllers operate at high frequencies. Timing is crucial for those systems. Timing behavior of software on real hardware needs to be tested before installation on a vehicle. This is where hardware-in-the-loop tests come into the play [12] [13].

Hardware-in-the-loop is a widely used testing platform for validation and verification of the ECUs. It is irreplaceable when running tests on real systems might be very expensive (e.g. hard to find errors, expensive to fix faults), damaging (e.g. easy to damage engine due to faulty software), dangerous (e.g. autonomous drive systems, flight controllers), or even impossible (e.g. space vehicles). The HIL test benches are mostly used in the aerospace, automotive and rail industry. Due to confidentiality and in-house development of early implementations, it is hard to track down historical records of HIL systems. The first HIL system was created in 1910. It was a flight simulator designed to protect human life as well as the machine itself. Physical controls were placed in a mockup cockpit and the effects of wind on the flight were simulated. Wind had to be real. After advances of computers starting from the 1950s and onward, software-based simulations replaced physical simulations. The magic of HIL systems is their ability to bring together the real world and virtual world. The first HIL systems used in the automotive industry were driving simulators. After the 1990s, commercial HIL systems were used in every industry where safety-critical aspects need to be tested. Steer-by-wire and brake-by-wire systems are examples of safety critical systems. The purpose of functional safety as defined in ISO 26262 automotive functional safety standard is to minimize danger in the event of a failure [14] [15] [16] [17].

In industries such as nuclear power plants, railways, aerospace and automotive creating a prototype and doing tests on that is a very expensive process. It is very hard to find faults on-vehicle or system bench tests. Even if they are found closer to the top of the V-model the more expensive it is to fix faults. There must be a way to do component integration tests before they can be installed on the final product. These industries are real benefactors of the HIL systems. Controllers on those systems control mechanical parts such as engines. If a controller is directly put on an engine without prior testing, engine damage is unavoidable. Damage further increases development costs. Autonomous vehicle driving tests, or airplane flight tests with real machines might be dangerous to the machine itself and human life. Finally, for some vehicles such as the space shuttle actual vehicle tests might be impossible.

Rest-bus simulation can replace hardware with its software equivalents in real time. Missing CAN nodes can be simulated in a virtual environment. From the perspective of a physical controller, those virtual nodes are undifferentiable from actual ones. Apart from previously aforementioned advantages, HIL simulation has additional benefits. A few of those benefits worth mentioning are rapid control prototyping and automated testing.

In summary HIL systems are an indispensable part of ECU testing and validation. HIL simulators have hardly any drawbacks and currently have no alternative to replace them. These platforms are expensive, and their software toolset is complex to master. HIL testing platforms have long queues, developers waiting their turn in lines. Due to ever increasing software size and testing frequency, HIL system tests are becoming a bottleneck for software developers. Testing time and testing cost of the HIL systems can be reduced further by introducing frontloading and this will be the motivation of this thesis. Alternative HIL systems can be built using off-the-shelf hardware and software on every developer's desktop as an addition to established HIL platforms.

Chapter 2 is going to provide technical background to the subject. A detailed investigation of state-of-the-art solutions will be done in Chapter 3. A series of experiments as well as proof-of-concept applications will be demonstrated in Chapter 4. Finally, in Chapter 5 and 6 results will be evaluated, and a conclusion will be drawn.

2 Technical Background

2.1 Software Testing

The goal of testing is to make sure that the software is running as intended. To do that errors and faults in the software need to be found, tracked, documented, and fixed. The cycle of test-find-fix needs to be repeated. Testing all combinations of inputs and outputs of a system is practically impossible because it is an infinite task. But there is a possibility to eliminate most of them. Suppose there is an eight-character long password with alphanumeric values as an input of the system. The number of different combinations reaches trillions for such a simple test. Besides, these are only for valid input values. A system also needs to be tested for invalid values. Testing for valid and invalid values expands test coverage. ISO 26262-6 is a software functional safety standard designed for the automotive industry and provides requirements for design, implementation, verification, and validation. It is a variation of the IEC 61508 standard used for commercial vehicles. The standard lists development phases and techniques to be used in each stage and only gives recommendations, and doesn't specify any preferred implementation. Techniques listed as "Highly Recommended" must be implemented, "Recommended" ones should be implemented and others are optional. Standard also encourages using common software development tools such as MATLAB and Simulink in embedded systems. The whole right-hand side of the V-model is dedicated to testing. HIL, integration, and vehicle tests are highly recommended by the standard for an ASIL (Automotive Safety Integrity Level) D system. Electric steering wheel is a typical ASIL D system found in a car, while cruise control is classified as an ASIL A system. ASIL classifications are done based on how much injury and harm it can cause in the event of a failure. Exposure (E), Controllability (C) and Severity (S) are the parameters used for classification. The standard requires a safe software development process must implement the V-model [15] [18].

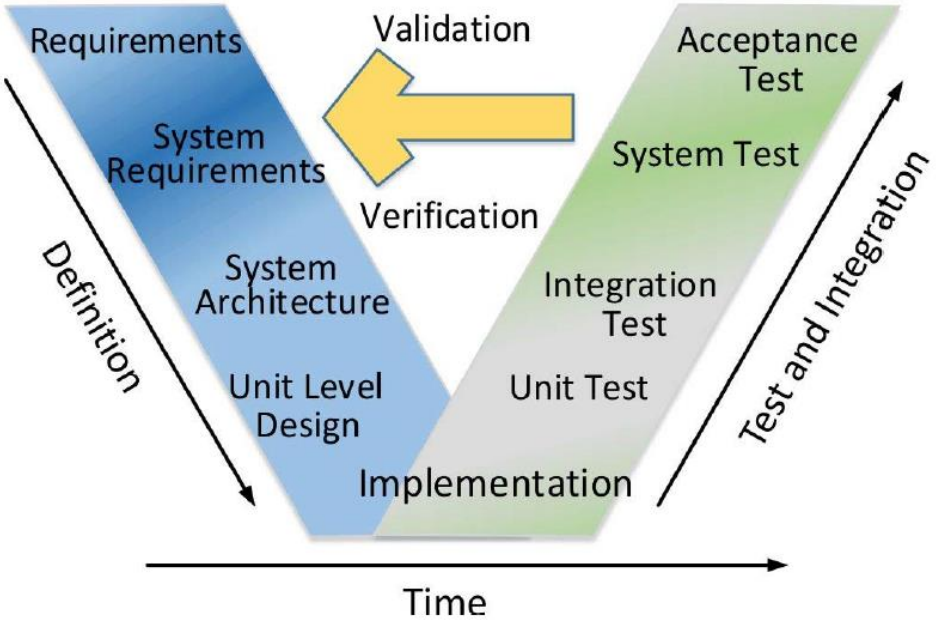

Figure 3 V-model of functional safety for software development process [15] [19].

Testing starts from the bottom of the V-model where models and software unit tests are done to the top of the V-model where acceptance tests are done. Road testing is the last step before acceptance testing is done by the client. Historically road testing was abused to find software faults. Instead, it should have been used only for calibration and validation. Finding software defects on-vehicle can be difficult, even impossible. Even though it can be used to find some faults, total time spent finding isn't worth the effort. Faults found closer to the top of the V-model are more costly to fix compared to faults found on the lower levels of V-Model [20].

Software unit tests are done by developers. The software system is built by combining all components. System tests can be done on the host platform or target platform. If tests are done in the host environment it is a SIL test. Host tests are done before actual hardware tests because it is easier to do simulation. If software is put on target hardware (ECU) then it is called a HIL test.

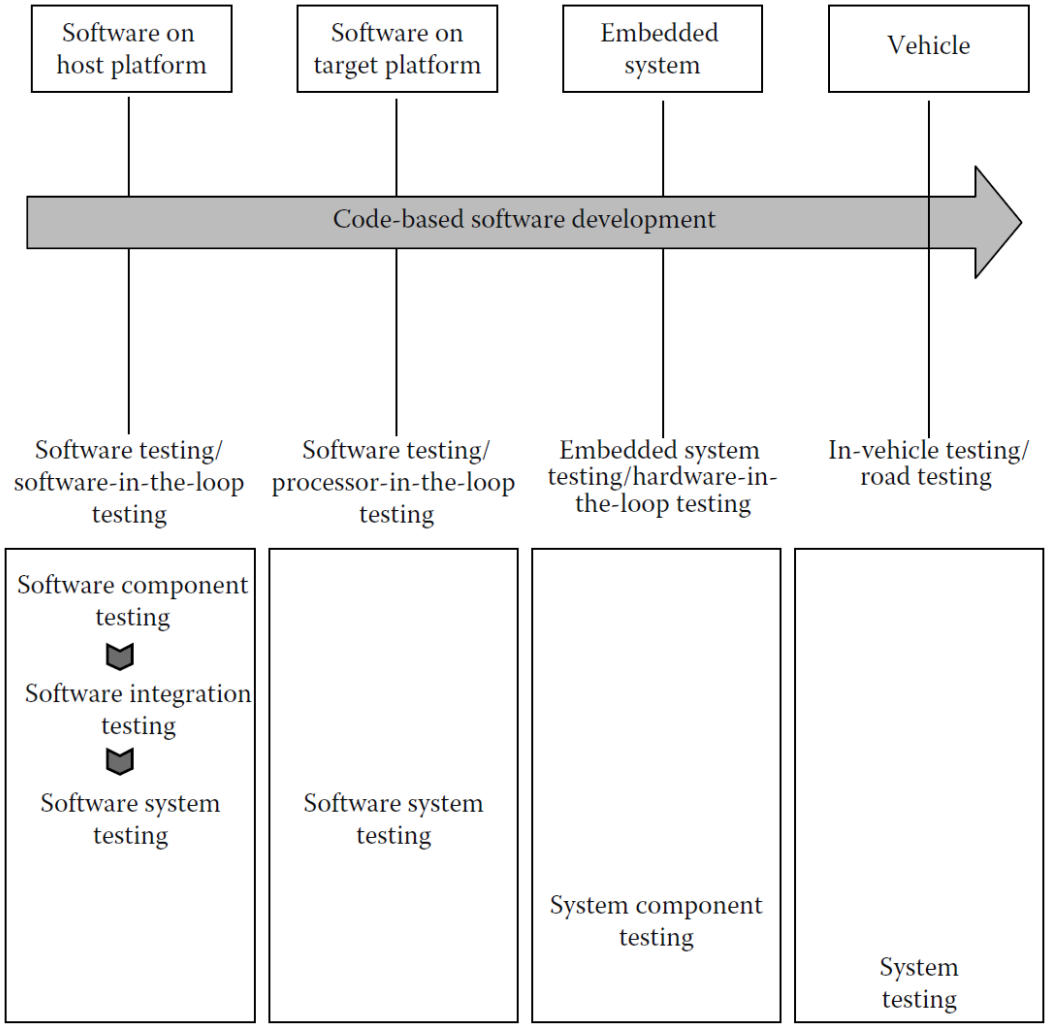

Figure 4 Software testing levels [5].

There are two types of analysis that can be done on the software. Static analysis is done on source level. Code review, inspection, walk-through are some common activities done at the static analysis stage, but actual code is not executed. Recently new techniques have been suggested for static analysis. Inclusion of AUTOSAR configuration files to the static analysis phase has potential to eliminate improperly prepared config files [21] [22]. Dynamic analysis on the hand requires program execution to check failures. Input space of variables can become infinite even for a simple test case, so a representative of input values set must be chosen to test the program. Equivalence partitioning and boundary value analysis are among those techniques used to determine representative input values [18] [20].

Verification and validation are two concepts frequently used in software testing activities. Verification is an activity evaluating the software at the end of a given phase against its specifications established in the beginning of the phase. Validation on the other hand evaluates software against initial customer requirements. In other words, it

examines whether software satisfies the customer's expectations. Late validation activities are risky and may end in higher development costs. One way to avoid late validation is converting to the agile methodologies. Agile methodologies interact with the customer in the early stages of development, doing verification and validation tests iteratively [20] [18].

White-box testing aims at structural application internals. Problems such as control structures and logic errors as well as execution paths, logic conditions, boundary of loops, data structures are found under this test. White-box tests focus on units where methods are tested individually. The other tests that are done under the white-box are fault insertion, memory-leak and coverage analysis. Black-box tests focus on application externals. Only public interfaces, inputs and outputs of application are tested against requirements. Incorrect functionality, missing functionality, interface faults, usability errors, performance problems, etc. are the types of problems found under black-box testing. Some of the black-box tests are functional tests, stress tests, performance tests, boundary tests, system tests etc. [18] [20] [23].

Equivalence partitioning is a test technique done under black-box testing. Building all possible input values is practically impossible, however building some representative subsets is possible. Partitions represent input sets that undergo different treatment. For a given partition test object behaves either correctly or erroneously for all values of a given partition class. Boundary value analysis uses input values that lie on the endpoints as minimum and maximum values. Analysis also includes representative values inside the boundary and outside of the boundary. Out of boundary values are named invalid values [5] [18] [20].

System testing is a synonym used for black-box testing. The reason behind this is that testing only concerns with application's externals. System testing is an umbrella term used for several tests. Some of those are functional, regression, security, stress, performance, among others. Functional tests validate the system against end user requirements. They target the business logic of the software. Once initial time and effort is spent on preparing testing scripts, it is very easy to replay them when a new functionality is added to the system. Scripts can be scheduled to run any time even at off times. Smoke tests are known as build verification tests. These tests are done after each build and are a subset of all tests that contain the most important functionality of the system. Fixing one defect may introduce another defect in other parts of the software. Regression tests are done when a new functionality is added to an already tested system. Regression tests are done to make sure new functionality doesn't break

existing functionality. Regression tests are repeated, more involved, and time consuming compared to smoke tests and hence are great candidates for automation [18] [20] [24].

Performance, scalability, security, concurrency, memory leak tests on the other hand constitute non-functional tests. Stress tests are designed to evaluate system performance under heavy load. Multiple users interact with the system simultaneously and the system is enforced to process massive amounts of data. By monitoring the system under heavy load, the test engineer wants to find the weakest point of the system, the point where it breaks first. Stress test scripts may be reused for performance testing also. Performance testing focuses on application performance criteria. Response time of the system can be measured and documented as performance indication. Aside from response time, disk usage, memory usage and CPU usages are also monitored. Automation is again very helpful for creating fake users. The number of virtual users is limited by computer resources. Acceptance tests are done by customers. Involving customers at early stages of development is beneficial than doing it at later stages. After completion of system tests acceptance tests start. Commercial software usually does not include acceptance tests. Instead, they deliver products to a large number of users as alpha and beta versions. Once faults are discovered by end users early releases, they can be fixed afterwards [18] [20] [24].

Customers demand quality software in a fast and at the same time cheapest way. This is the biggest challenge of software testing. Whoever delivers a quality product to the market first wins. The size of the software reached up to millions of lines. The cost of software errors to a nation's economy can reach up to billions of dollars. Cost and time spent on testing takes almost half of the overall project's cost and time. Therefore, improvements made in testing time have valuable consequences. Time savings can be calculated by Eq. (1) [23].

$$\begin{aligned} Total\ test\ execution\ time\ &= Planned\ tests\ count \\ &* Estimated\ time\ to\ execute\ one\ test \\ &* Test\ iterations\ count \end{aligned} \tag{1}$$

Not all benefits can be measured in terms of man-hours. In those cases surveys can be used to measure benefits [20]. AST (Automated software testing) life cycle contains test planning, test script development, test data generation, test execution, result analysis, error monitoring and report generation. To summarize, automation reduces

time and cost of testing, also improves software quality by increasing coverage by replacing error prone and labor-intensive manual testing with more data variations, more test scenarios, more paths, and branches. Finally, keep in mind that some tests can't be done manually such as memory leak detection, concurrency testing, performance testing etc. [23].

What is software quality? The most influential quality model is proposed by ISO 9126 Software Engineering product quality [25]. The document describes six characteristics of the quality. These are functionality, reliability, usability, efficiency, maintainability, and portability. Verification and validation tests clear faults and defects result in increased functionality which in turn increases quality. Software quality according to definition found in ISO is a measure of a system how accurately it satisfies all user requirements (functional requirements) and how reliable it is under non-functional requirements (stress, concurrency, security, etc.). The metrics associated with software quality are defects density, MFFT (mean time to failure), MTTCF (mean time to critical failure), number of high priority defects, reliability (probability that no failure will occur in next n-time interval) [18] [23]:

No matter how many tests are done there will always be some defects undiscovered, and those defects will eventually surface at the customer site. Those defects will cause the system to fail from time to time. The failure rate is measurable and decreases as faults are discovered and fixed. The objective of testing is to reduce failure rate to an acceptable level. Therefore, reliability is a quantitative measure useful to address quality of the software. AST results in more testing for a given time frame. As testing continues, we can expect that fewer defects will be discovered at following iterations. Figure 5 shows the relation between testing time and the percentage of undiscovered defects.

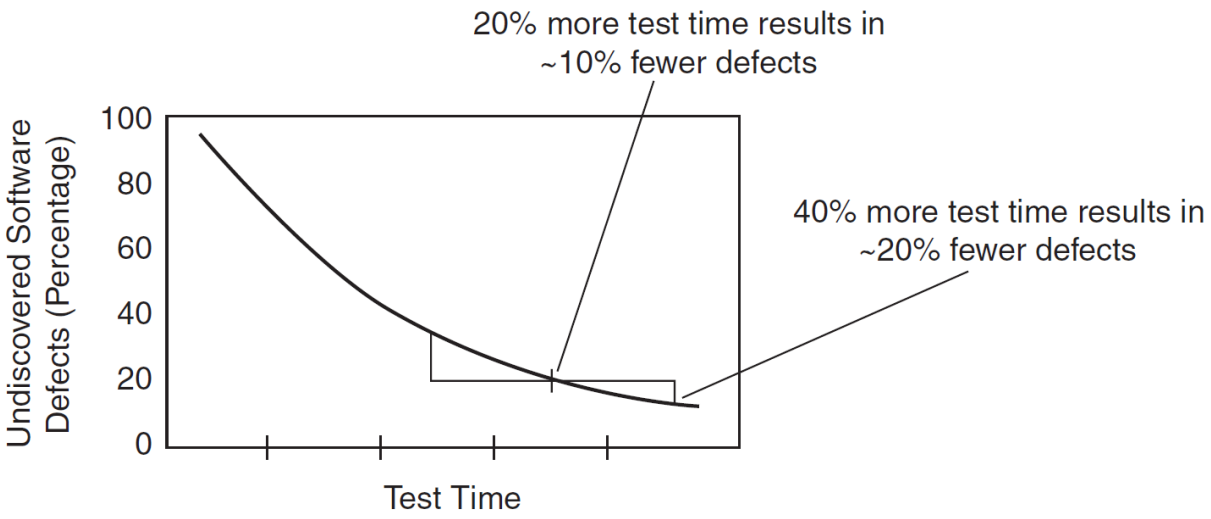

Figure 5 Relationship between testing time and percentage of undiscovered defects [23].

AST increases coverage by executing thousands of tests at the same time. Expanded coverage uncovers some hidden and tricky defects that nobody expected to be there. Software products that don't function most of the time are considered unreliable. As testing time increases defect density decreases, reliability and hence the quality increases. The quality of software perceived by the customer is different from the quality of software calculated by defect density. For example, ten defects slipped into the actual production version occurring frequently and affecting critical functionality is perceived as poor-quality software by the customers. On the other hand, 100 defects occur less frequently, and affect non-critical functionalities perceived as good quality software by the customers [23] [24].

2.2 Test Automation

Automated tests have many benefits compared to manual ones. Automated software testing (AST) means to replace manual tests done at any phase with test scripts executed by computers. Test scripts are written in a programming language and can be prepared by engineers or can be auto generated. In other words, a software is developed to test another software. Automation isn't intended to replace all manual tasks done by humans. Because not every task is suitable for automation. Some tasks are impossible to automate while others don't pay off the investment put on to prepare test scripts. AST and manual software testing complement each other. A report generated once a year and needs a lot of time to prepare a script has low ROI (return-on-investment) metric and isn't an ideal candidate for automation. Repetitive labor-intensive and less changing tasks are the ideal tasks that are best to be converted to automated ones.

The primary objective of automated testing is to reduce time and effort spent on manual testing. Other benefits are increased coverage, reduced testing costs, and increased productivity. Productivity is increased because automation liberates programmers from doing mundane repetitive time-consuming tasks. Doing tests manually incorporates human errors. Repetitive and time-consuming tests create lots of fatigue in testers. Automated removes human factors from testing procedures and hence increases reliability. Another advantage of test scripts is that they can be played back at any time. This way once an error is found the same erroneous condition can be re-created by executing the test script again. Repeatability of erroneous conditions increases software quality. When testing is done manually, steps leading to a faulty condition may not be remembered for the second time. Some tests can be difficult and inaccurate when done manually. Tests require interaction with software in milliseconds precision can't be done manually at all. Likewise, tests requiring parallel interaction of hundreds or even thousands of real users are difficult to do. Automated testing tools have load simulators to overcome many user problems. Cost also increases as the number of working hours and resources are put on tests. Reducing manual testing, reduces expenses spent on people and testing equipment [20] [23] [24].

Automatic generation of test scripts instead of manual preparation is the idea behind MBT (Model-based testing). Test scripts are generated based on the SUT model. This makes test scripts linked directly to SUT and hence user requirements. MBT complements model-based development (MBD) has direct relation to SUT model and user requirements [26].

With only the press of a button test scripts can be executed. Beauty of automation is its scheduling flexibility and parallelism. Tests can be scheduled to run after-hours or can be triggered whenever a change occurs in a software. Either all scripts or some selection of them can be started at a time. In the smoke tests for example only a high-risk subset of all tests is executed. Executing test scripts anytime and in parallel reduces testing time considerably. A study shows that automation consumes only 25% of the manual testing time [20].

CANoe is equipped with a Test Feature Set. Automatic testing can be started only while measurement is running. Test modules can be written either in CAPL or .NET language or can be formulated as an XML file. Each module contains individual test cases. Test cases contain actual instructions and should be structured as independent units. Each test case inspects some part of the system. Test cases contain many test steps. A step results in either a pass or a failure. Execution of a test is interrupted only

when “waitXXX” commands are used. While waiting, the control is transferred back to simulation. As soon as the waiting event returns, control is transferred back to the test module again [27].

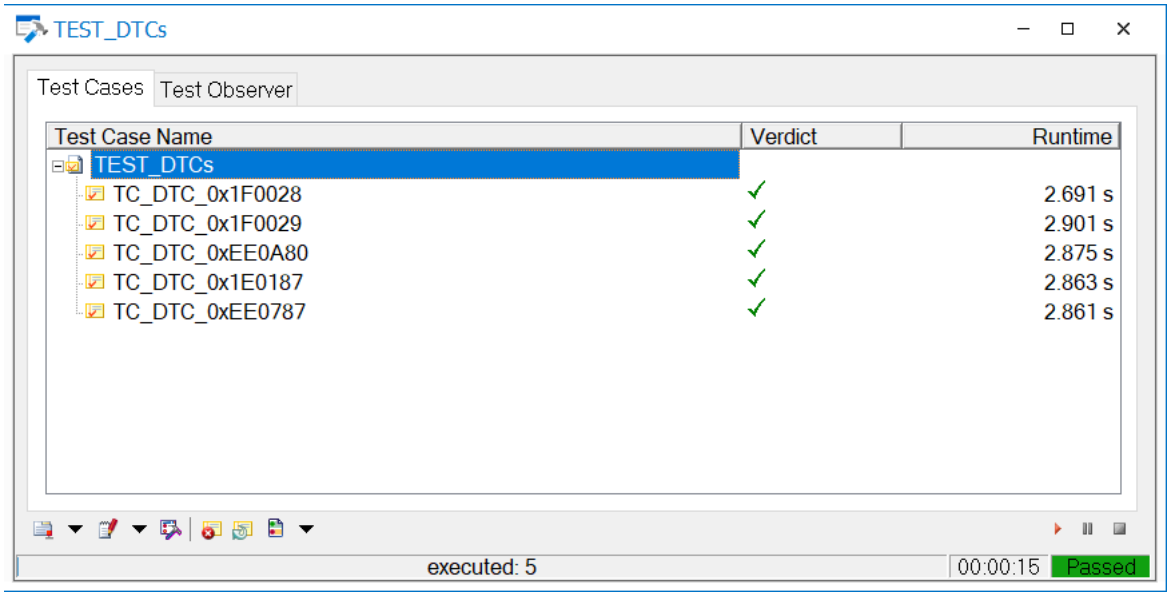

Figure 6 Test module execution.

After execution of test cases of the module, results need to be evaluated. If the expected results and the outcome of the tests are equal, then the verdict is a pass, otherwise it is a failure. Figure 6 shows the TEST_DTCs module and its executed test cases. According to the figure, all test cases are passed, and a total of 15 seconds is spent on the execution. Many automations software has built-in reporting tools. Reporting tools assign green color to the passed test cases while red color to the failed test cases. Reporting tools eases error analysis considerably by identifying failed scripts, test cases and obtained results. However, not all failures are defects. There might be false positives present in the report. False positives might be a result of user errors, environment errors or setup errors. After test execution optionally an HTML report can be generated. Figure 7 is the report generated for the TEST_DTCs module after its execution. Since all test cases are passed, the report shows us 100% pass result. In case there were failed ones, we would see them in red color with additional information printed by test cases. Test cases use library functions to print out additional information to the test report.

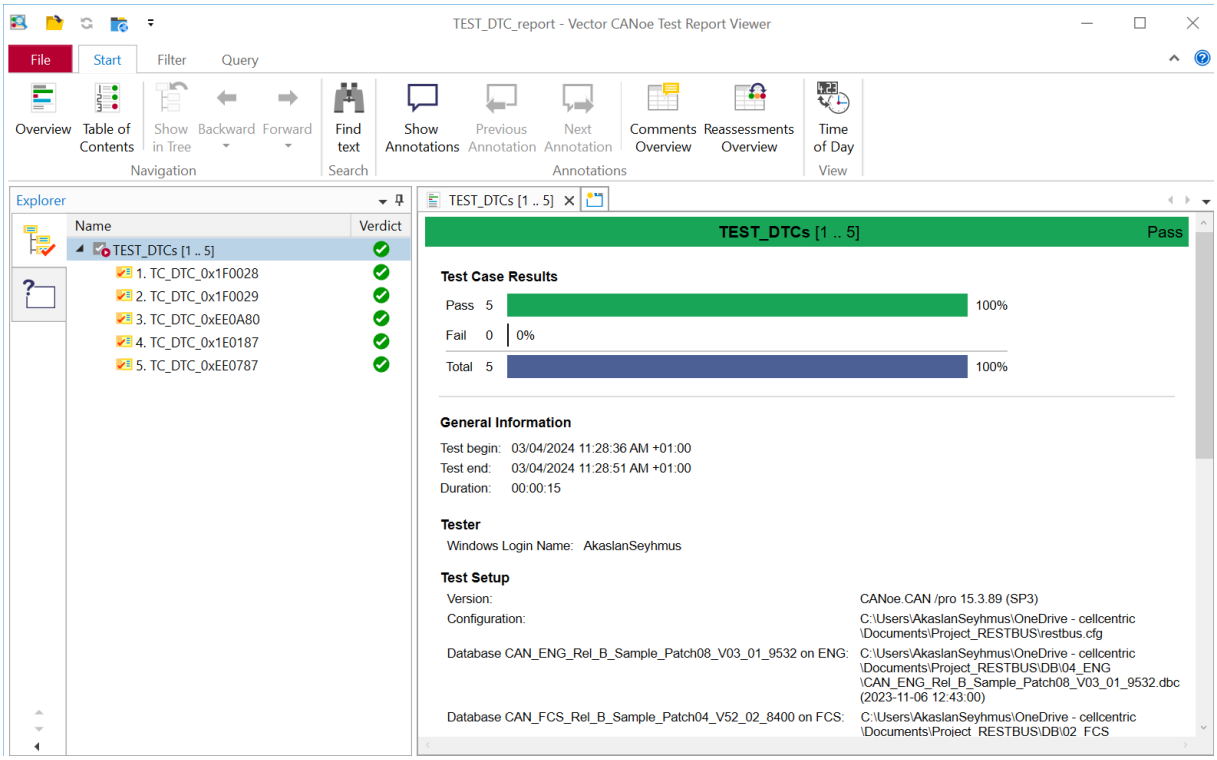

Figure 7 A test report is generated after module execution.

Once failed test cases are determined, corresponding defects afterwards need to be tracked. Defects are classified based on priority levels such as level 1, 2 and 3. L1 represents highest-priority defects and L3 represents lowest priority defects. Acceptance criteria should state which levels of software defects should be resolved. As soon as all defects are fixed based on acceptance criteria then software is ready for acceptance testing [20] [23].

2.3 Simulation and Testing Tools

CANoe/CANalyzer is a tool used for testing, development, simulation and analyzing ECUs in different bus networks including CAN, CAN FD, LIN, FlexRay, Ethernet, etc. The difference between CANoe and CANalyzer is that the former supports simulation and automation of tests while the latter doesn't. This thesis will cover almost all aspects of the CANoe tool. CAPL is the programming language used internally by these tools. CAPL scripting allows easy scripting for simulation and testing functionality. CANoe can also be integrated with MATLAB/Simulink models as well as other applications through Microsoft COM interface. CANalyzer acts as a single node, whereas CANoe supports unlimited nodes. Measurement starts only when the user presses the lightning bolt icon residing on the top left toolbar. A measurement is a series of activities such as reading messages from channels, processing them, and outputting them on output channels.

Analysis is supported primarily by trace window, graphics window, logging capabilities. These tools reside in the measurement window of CANoe. Trace Window displays flowing messages together with their meaningful symbolic names. Resolution of symbolic names requires a database file attached to the analyzed network. Supported database files are DBC file CAN networks, LDF file for LIN networks and FIBEX files for FlexRay networks.

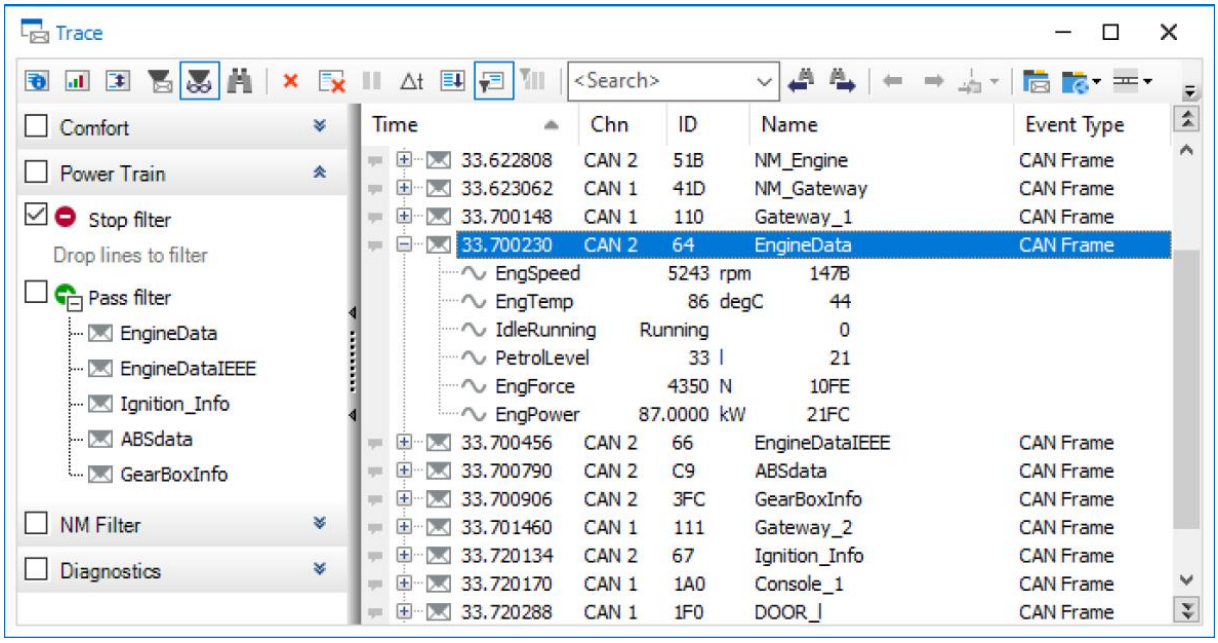

Figure 8 Trace window of CANoe [28].

Trace window supports message filtering by either using a stop filter or a pass filter. Analysis is done by observing messages and their signals. Messages further can be filtered before reaching the trace window by using programmed filter blocks. Filters reduce the amount of data displayed or logged to the corresponding analysis windows.

The graphics window depicts progression of signal values, system variables and even diagnostic parameters with respect to the time. The window has built-in features such as zooming, trimming, setting markers and statistics calculations. Statistical window computes and shows statistical values such as min, max, average, and standard deviation for each signal. Figure 9 depicts a graphics window of CANoe with a measurement of signal with 20ms period on CAPL-on-board mode.

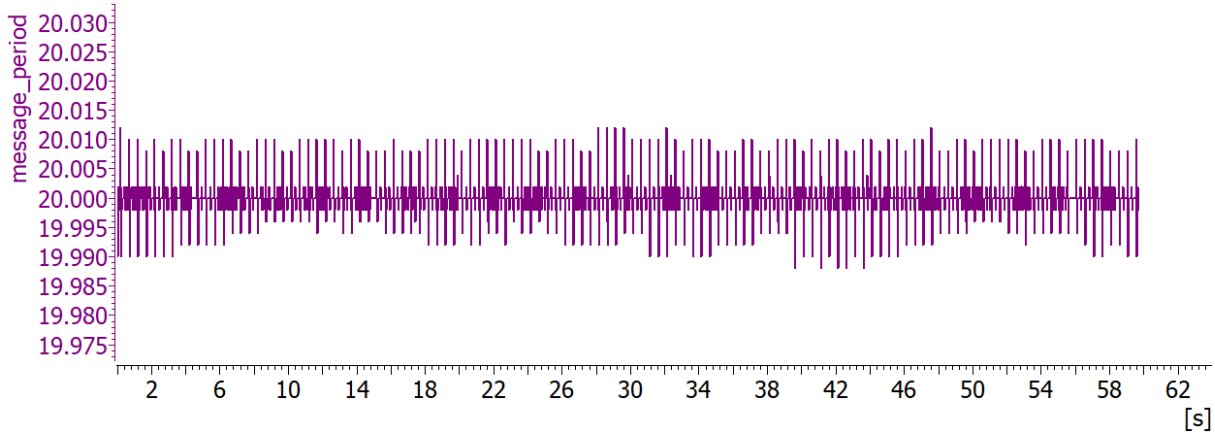

Figure 9 CANoe Graphics window showcasing 20ms periodic message on CAPL-on-board.

CAPL is a proprietary language used internally by CANoe. It is similar to C-language but provides an event-driven programming environment. Some of the supported events are keyboard, system variables, messages, timers, and diagnostic messages. A typical CAPL program consists of event blocks. In the absence of events, CAPL programs stay idle. CANdb++ is a software tool used to create CAN database files. Database stores information related to signals, messages, and nodes. Database files are easy to pass around to other colleagues and hence increase collaboration. Putting those definitions in a single file promotes consistency and prevents errors. CAPL browser can also use database information to access symbolic names of messages and their signals. CAPL Browser is used for coding the actual scripts. The CAPL browser supports autocomplete and symbolic definition of messages when a database is attached. Simulation and testing are the primary use cases of CAPL scripting. This browser comes with many features such as integrated symbol explorer gives easy access to system variables, database symbols, test functions, event blocks etc. Figure 10 shows a CAPL browser [29].

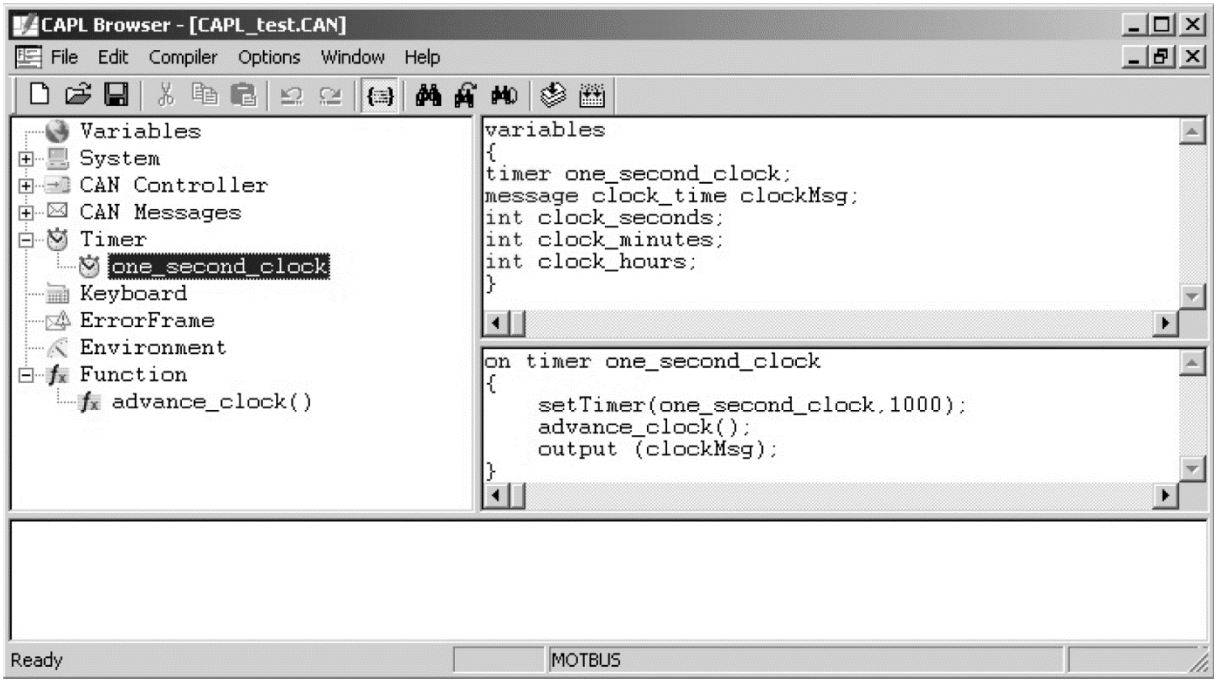

Figure 10 CAPL Browser screen [30].

Write window is used for displaying outputs of scripts. CAPL programs write to the console by “write()” API method call and send messages to the bus with “output()” method call. Debugger support is only available in simulation mode because debugger might need to stop execution of the script. In the presence of real hardware, debugging is supported only in test modules.

A common approach used by engineers is to use CANoe in a 3-phase development stage. In the first phase all nodes are simulated in the CANoe environment. This stage is used by architects to test the overall system’s feasibility. In the second phase, over the course of development some nodes will be replaced with real ones. This stage is the merging of the real world with the virtual one. Virtual nodes serve as rest-bus simulation. In the last phase all nodes are real, and CANoe is used for analysis and checks integration of all ECUs [28] [31]. All stages are depicted in Figure 11.

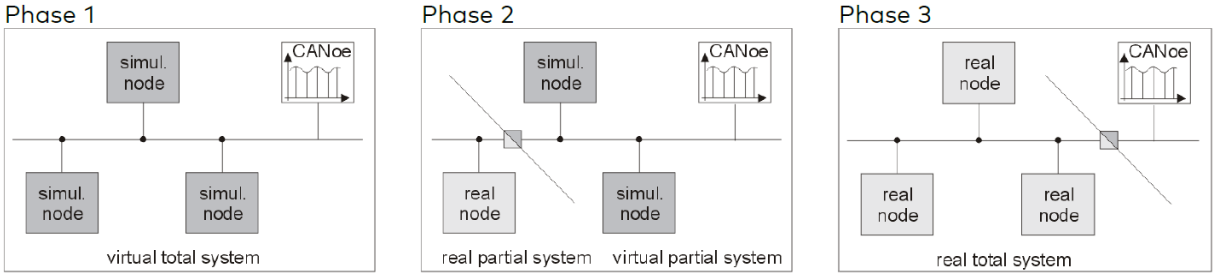

Figure 11 Common development stages in CANoe [28].

Panels are created programmatically and serve as the virtual cockpit of the system. Panels provide developers with common screen controls such as text and number fields to do either input or output. Panels are wired to system variables behind the scenes. System variables make connections between CAPL programs and associated panels. On the CAPL side, the program can react to changes of system variables and handle them in event blocks.

Generators are used to send signals either manually or periodically. A list of predefined waveforms can be attached to signals. CAPL knowledge is not mandatory to use a generator. Apart from choosing a wave form, a raw or scaled physical value can also be set individually for each signal. Physical values are related to raw values by the scale equation.

$$\textbf{physical value} = \textbf{raw value} * \textbf{factor} + \textbf{offset} \quad \textbf{(2)}$$

Raw value is multiplied by a factor and added an offset to get the physical value. Physical value is generally accompanied by an optional unit and min, max limits.

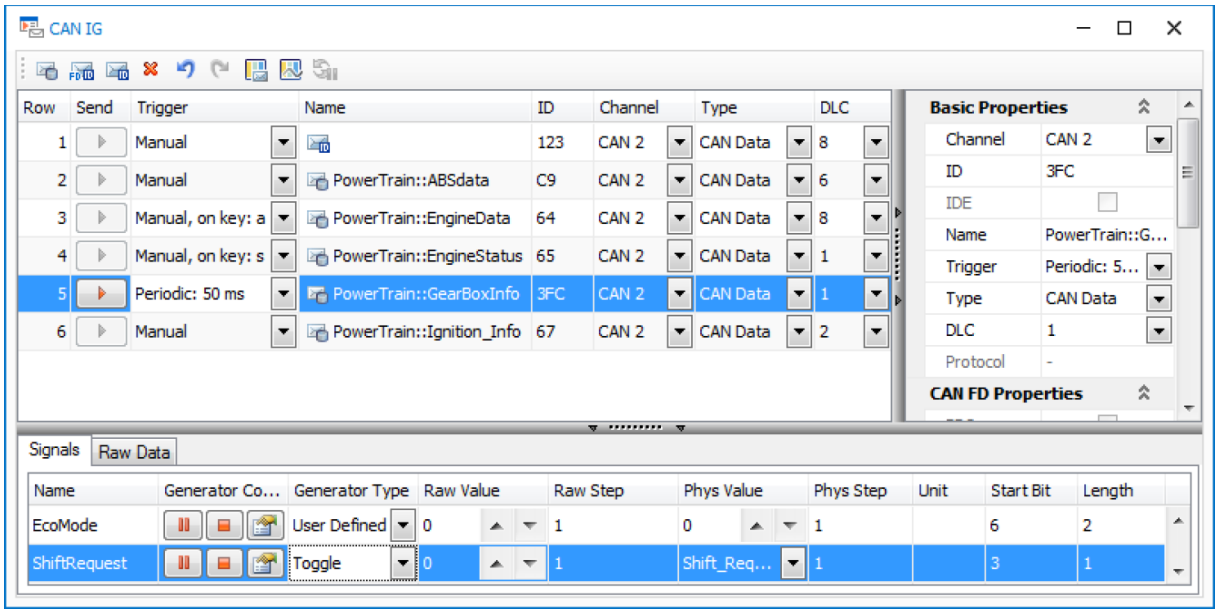

Figure 12 Interactive Generator (IG) window in CANoe [28].

2.4 CAN Communication

CAN was first introduced by the Bosch company in 1983 to replace point-to-point serial communication protocols used in the early days of the automotive industry. After that it was standardized by ISO 11868 in 1993. Main objectives were reducing complexity arising from point-to-point (P2P) networks and reducing the length of wiring harness. The wiring harness is reduced due to replacing P2P cables with a single twisted two-wire bus cable. Today CAN has become the most widely used multi-master serial

network in automotive. CAN is an CSMA/CA (Carrier sense multiple access collision avoidance) protocol and event triggered. High-speed CAN operates in the range 512Kbps-1Mbps. On the other hand, in time-triggered communication each node gets a time slot/slice to transfer its messages in a periodic manner. TT (time-triggered) networks are preferred for safety critical applications such as X-by-wire applications due to their deterministic nature. FlexRay is a commonly used TT network in automotive and operates at speeds between 10-20Mbps. CAN communication is predictable, but it is not deterministic due to message priorities. CSMA protocols check medium status to determine whether it is idle or busy before engaging in a transfer. Transfer happens only when the medium is in an idle state. This behavior constitutes a carrier sense part of the protocol. Arbitration mechanism resolves conflicts when two messages are transferred at the same time [5] [31].

CAN transfer its messages using frames at speeds between 10Kbps up to 1Mbps. The frame contains various fields such as ID (identifier), control, payload, and CRC. CAN doesn’t need any configuration to function. That means any node can be added to the network anytime without configuration. Each message has an associated ID field. Priority is determined by ID. The lower IDs have higher priority. This happens due to logical zero corresponding to dominant state and logical one corresponds to recessive state. Classic CAN comes in two flavors. The first one is standard CAN which has an 11-bits ID field and other one is extended CAN which has 29-bit ID field. Each node can filter messages it is interested in. Data consistency is satisfied by using error frames. Either all nodes accept a message or none of them [32] [33].

Safety of a CAN message is achieved by error detection mechanisms. There are five error checking mechanisms built in the CAN. These are bit errors, stuffing errors, CRC errors, ACK errors and form errors. Once an error is detected by any node, the error frame is transmitted, the related message is discarded by receiver nodes and retransmitted by transmitter node. Bit errors are detected by transmitter nodes. Transmitter node monitors compare sent bits with actual value on the wire. In case they are different then an error flag is transmitted. Differences in the arbitration field, ACK field slot and passive error flag are exceptions. Stuff error is detected when any node monitors six consecutive identical bits. CRC error is detected by receiver nodes if calculated CRC differs from transmitter node. Error flag is transmitted after ACK delimiter. CRC delimiter and ACK delimiter always need to be recessive. Form errors happen when a violation occurs in those fields. ACK errors are detected by the transmitter node when the ACK field remains recessive. This field is supposed to be filled by receiver nodes [32] [5] [33].

CAN has four different frame types. Data frame is used for transmission of a payload. The remote frame itself doesn't carry any payload but is used for requesting data. Error frames are used to flag an erroneous message and finally overload frames are used to insert delays between successive messages. The frame format of data and remote frames are similar to each other. Likewise frame format of error and overload frames are similar to each other. Fields of standard format are start of frame field, arbitration field, control field, data field, CRC field, ACK field and end of frame field. These fields are depicted in Figure 13.

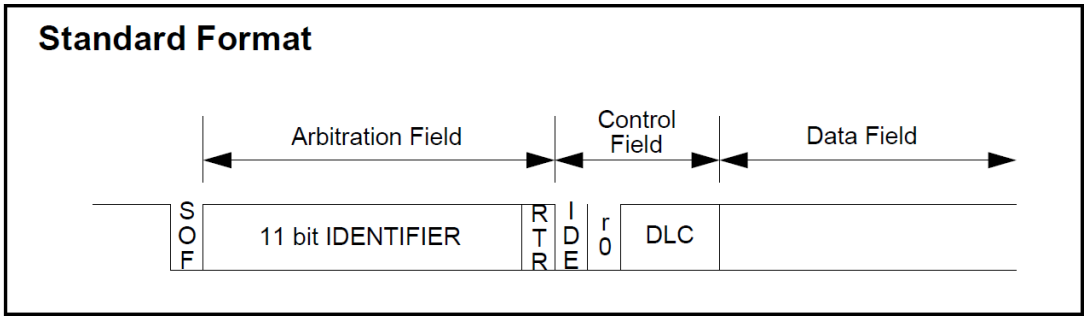

Figure 13 CAN Standard frame format [32].

In the standard frame also known as the CBFF (Classic Base Frame Format) arbitration field is composed of an ID field and RTR bit. In extended cases arbitration is composed of SRR bit, IDE bit and RTR bit. The IDE bit is the marker that distinguishes extended frames from standard ones [32] [33]. Extended frame format also known as CEFF (Classic Extended Frame Format) is given in Figure 14.

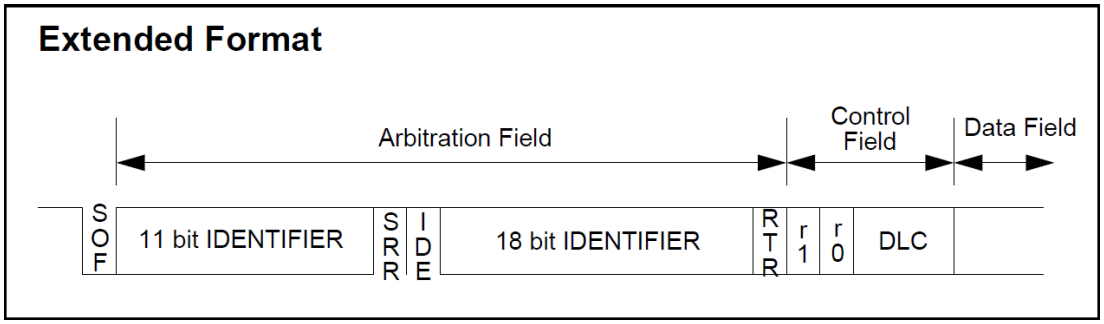

Figure 14 CAN Extended frame format [21].

Bit stuffing covers the start of frame, arbitration field, control field, data field and CRC field. The remaining fields as well as CRC delimiter aren't stuffed. Whenever five identical consecutive bits are detected in the bit stream, the transceiver adds a complementary bit into the stream. On the receiver side, stuffed bits are removed.

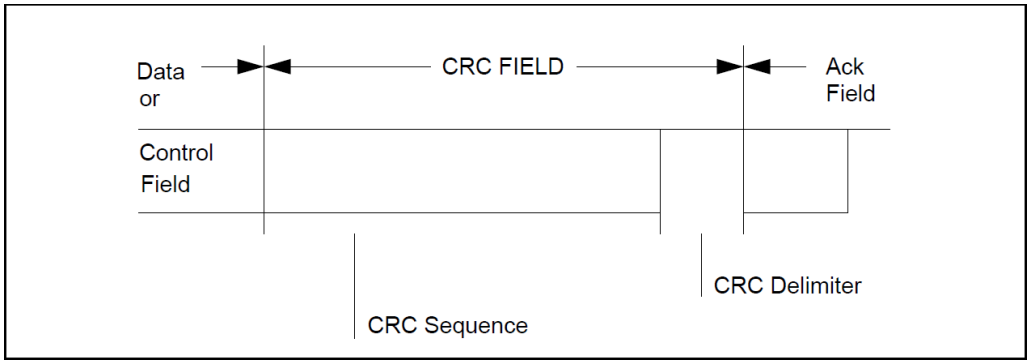

Figure 15 CAN frame CRC field [21].

There are two types of error flags depending on error counters of nodes, either error passive or error active flag is sent. Six consecutive dominant bits are sent by error active nodes, on the contrary six recessive bits are sent by error passive. If an error frame destroys bits between start of frame and the CRC delimiter, it is interpreted by other nodes as bit stuffing error due to violation of bit stuffing rules. Whereas if destroyed bits are ACK field and end of frame then other nodes interpret this as form error. Upon reception of an error flag, other nodes also send their own error flag [32]. This behavior causes error flag superposition which can be seen in Figure 16.

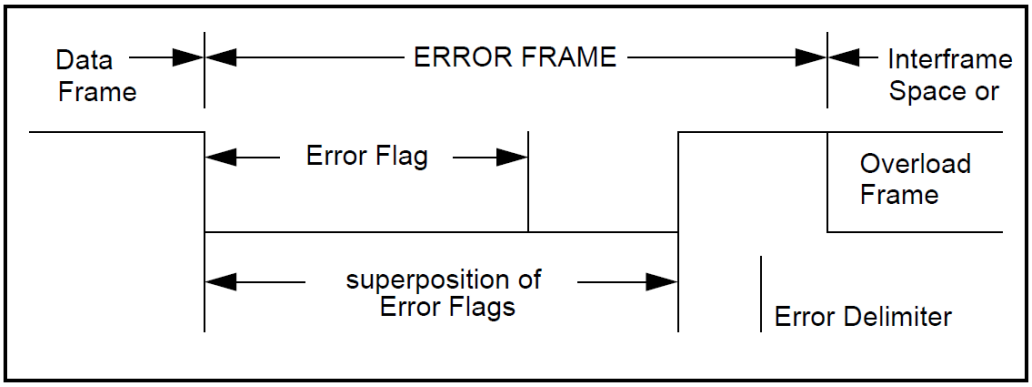

Figure 16 CAN Error frame format [32].

Equation (3) represents the worst case scenario when calculating the stuffed bits. The vertical bars are mathematical floor function operators.

$$ns(n) = \left\lfloor \frac{n - 1}{4} \right\rfloor \tag{3}$$

Bit fields from SOF to the CRC are subject to bit stuffing. However, bit fields after CRC, including CRC delimiter are not subject to bit stuffing.

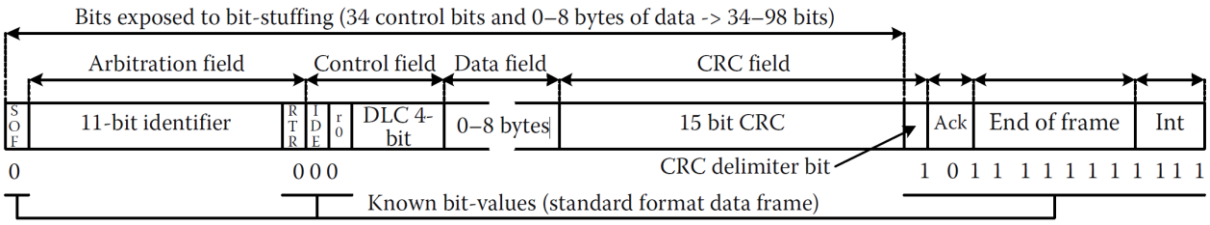

Figure 17 Bit stuffing for standard frame format [5].

Figure 17 shows all fields of a classic base frame which are subject to bit stuffing. The same information is true for classic extended frames too. In the worst-case scenario, the total number of calculated bits including stuffed ones in the classic base frame is 135 bits whereas in a classic extended frame it is 160 bits. These bits are transmitted under assumed 512Kbps network speed. Transmission time for an extended frame is $t_{classic\ extended\ frame} = \frac{160}{512} = 0.3125\text{ms}$ and transmission time for an classical base frame is $t_{classic\ base\ frame} = \frac{135}{512} = 0.2636\text{ms}$. The transmission time of an extended frame is approximately one third of a millisecond while base frame is one fourth of a millisecond in an empty CAN bus. To give an example of stuffed bits calculation following Table 1 lists all fields of an extended frame.

Table 1 Classic CAN extended frame stuff bits calculation.

Field Name	Size	Stuffed
SOF	1	1
Arbitration	12 + 20	32
Control	6	6
Data	8 * 8	64
CRC	15	15
CRC delimiter	1	0
ACK	1	0
ACK delimiter	1	0
EOF	7	0
Interframe	3	0
Total	131	118
Total Stuff Bits		$\left\lfloor \frac{118-1}{4} \right\rfloor = 29$
Grand Total	160	

By design CAN frames are subject to fixed priority scheduling due to arbitration favors lower IDs compared to higher ID numbers. CAN bus is non-preemptive and transmission time depends on presence of other frames. Transmission time is calculated using worst case response time (WCRT). The worst case is a critical instant

that happens when all frames are fired at the same time by all nodes. A low priority frame might block a higher priority frame at most for the duration of the message. A frame waiting for its turn has to wait until the busy period ends. CAN frame transmission time is predictable but it is not deterministic. Worst case response time is the longest time needed for a frame to finish its transmission [5]. WCRT formula in Equation (4) is an iterative equation where B_i is the maximum blocking time due to the lower priority frames, C_k is the transmission time for frame k and summation is over higher priority ones than the current one. Iteration is done until $t_i^{n+1} = t_i^n$ is satisfied [5]. If the calculated WCRT for all frames is less than or equal to the deadline of each frame, then it is said that the set is schedulable. If some frames miss deadlines, then those frames are not schedulable.

$$t_i^{n+1} = B_i + \sum_{\forall k \, \mathcal{E} \, hep(i)} \left\lceil \frac{t_i^n}{T_k} \right\rceil C_k \tag{4}$$

The fault confinement mechanism implemented in each CAN controller is based on two counters: transmitter error counter and receiver error counter. Depending on the values in these counters, nodes assume three states. Error active state allows nodes to participate in CAN and in an erroneous situation an error active frame is transmitted. The error passive state allows nodes to participate in CAN and in an erroneous situation an error passive frame is transmitted. The bus-off state doesn't allow nodes to participate in a bus. A node goes into a bus-off state if the transmit error count reaches 256 or above. WCRT calculation in Equation (4) doesn't account for error frames. Environments with high EMI backgrounds interfere with CAN bus and bit errors might occur. If error frames coincide with critical instant, then WCRT calculations need to include them.

CAN FD specification was introduced by Bosch in 2012 to further improve classic CAN. FD stands for Flexible Data-Rate. FD introduces faster communication speeds up to 5Mbps and increased payload up to 64 bytes. This is achieved by introducing a new frame format in the data-link layer while preserving the physical layer. FD frames are built based on either base frames or extended frames. FD base frame has an 11-bit ID field, whereas FD extended frame has 29-bit ID field. FBFF stands for FD base frame format while FEFF stands for FD extended frame format. CAN and CAN FD are interoperable as long as controllers adhere to Bosch CAN FD specification as well as CAN physical layer standard ISO 11898-1. This means FD controllers are backward compatible, can send and receive messages from CAN controllers [34] [33].

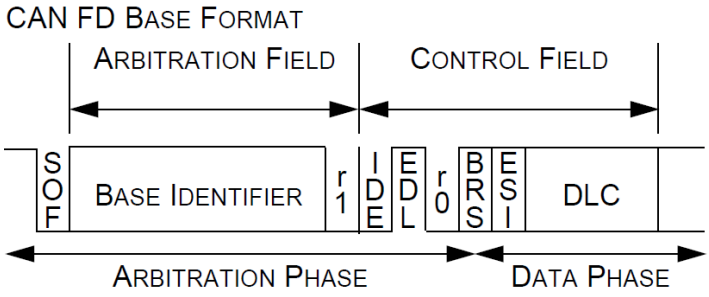

Figure 18 CAN FD base frame format [34].

The IDE (Identifier Extension) field is there to distinguish between base and extended frame formats even for FD frames. In base format it is always dominant whereas in the extended format it is recessive. SRR (Substitute Remote Request) transmitted only in extended frames and is recessive. It is introduced in extended frames to deliberately lose arbitration to the base frame in the case of collision. RTR (Remote Transmission Request) field only exists in standard frames, dominant in data frames and recessive in remote frames. FD frames don't contain remote requests. The R1 field is the replacement of RTR in FD frames. Its value is always dominant.

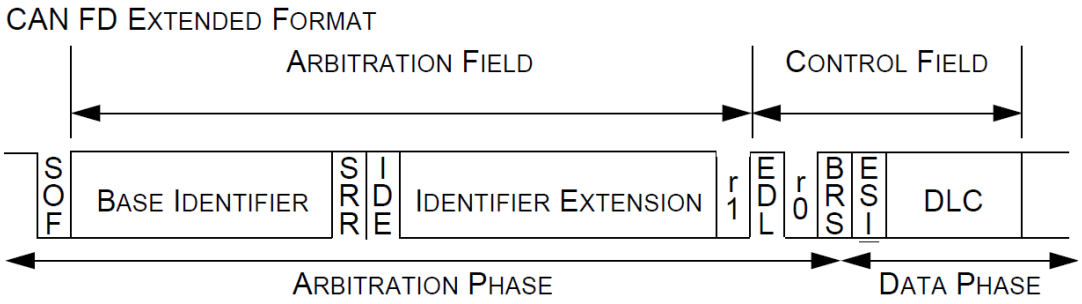

Figure 19 CAN FD extended frame format [34].

FD format introduces three new bit fields in the control field of the frame. These fields are EDL (Extended Data Length), BRS (Bit Rate Switch) and ESI (Error State Indicator). EDL is renamed as FDF (FD Format) in ISO standard. EDL bit field distinguishes FD frames from classic frames. Its place is filled with r0 in the classic frames and is always dominant. On the contrary in FD EDL is always recessive. The BRS field determines transmission speed. Recessive values mean switching to the higher bitrates whereas dominant value means low speed. Higher bit rates start at BRS bit and end with CRC delimiter. The ESI field is there to distinguish error active and passive nodes. It is dominant for error active nodes and recessive for error passive nodes. The interpretation of the DLC (Data Length Code) field is changed in FD frames to accommodate for 64 bytes payload. Depending on the length of data, CRC length also differs. CRC_15 is for 15-bits, CRC_17 is for 17-bits and CRC_21 is for 21-bits. FD frames don't include CRC fields into bit stuffing. Stuff bits for CRC fields are predetermined and fixed in FD frames. The other difference is the CRC delimiter field.

Table 2 CAN FD stuff bit calculation for extended frame.

Field Name	Size	Stuffed
SOF	1	1
Arbitration	12 + 20	32
Control	(9) 8	(9) 8
Data	8bit * 64 bytes	512
CRC	21 + 4	6 fixed stuff bits
CRC delimiter	1	0
ACK	1	0
ACK delimiter	1	0
EOF	7	0
Interframe	3	0
Total	566	553
Total Stuff Bits		$\left\lfloor \frac{553-1}{4} \right\rfloor + 6 = 144$
Grand Total	710	

Transmission time calculation for the FD frame is a little involved due to stuffing rules and different bit rates. For an FD network, bandwidth high speed is assumed to be 5Mbps and arbitration speed is 512Kbps. In the worst case the total number of bits transmitted in an FD extended frame is calculated to be 710 bits. Suppose the bit rate switch is turned on. First 36 bits up to BRS bit will be transmitted at low speeds. Bit fields between BRS and CRC delimiter will be transferred at high speeds. After CRC, low speeds will take over again. First $1 + 32 + 3 = 36$ bits with $\left\lfloor \frac{36-1}{4} \right\rfloor = 8$ stuff bits are transmitted under 512Kbps network speed in $t_{segment\,1} = \frac{36+8}{512} = 0.08594\text{ms}$. The second segment has $5 + 512 + 25 = 542$ bits and $\left\lfloor \frac{542-1}{4} \right\rfloor = 135$ stuff bits. For the second segment transmission time 5 Mbps network speed is calculated as $t_{segment\,2} = \frac{542+135}{5120} = 0.1322\text{ms}$. Lastly remaining $1 + 1 + 1 + 7 + 3 = 13$ bits will be transferred at 512Kbps network speed in $t_{segment\,3} = \frac{13}{512} = 0.0254\text{ms}$ time. Total time spent for transmission of an FEFF frame with BRS switch is on equals to following sum.

$$\begin{aligned} t_{total} &= t_{segment\,1} + t_{segment\,2} + t_{segment\,3} \\ &= 0.0859 + 0.1322 + 0.0254 = 0.2435\text{ms} \end{aligned} \quad (5)$$

The equation (5) tells us even though payload size increases from 8 bytes to 64 bytes

due to higher bitrates, transmission time for the FD frame is still less than comparable classic frames.

2.5 Diagnostics

Diagnostics is, as its name suggests, used to diagnose an ECU either in development phase or after its installation on a vehicle. Diagnostics works in a request and response fashion. Response can be either positive or negative. CANoe provides diagnostic functionality through diagnostic console and fault memory window. Built-in basic diagnostic editor allows defining services (UDS & KWP) for CAN, LIN, FlexRay and Ethernet networks. These tools rely on the transport protocol (TP) library and CAPL Callback Interface (CCI) implementations placed on the transport layer. TP library is responsible for segmentation, flow control whereas CCI is responsible for event mechanisms for diagnostic messages integrated in CAPL programming. To use diagnostics, diagnostic descriptions must be defined. These descriptions come in the form of CDD (CANdela Diagnostic Description), ODX (Open Diagnostic Data Exchange) or MDX (Multiplex Diagnostic Exchange) file formats. These files behave as a database of services and their parameters. After inclusion of diagnostic descriptions to the CANoe, additional event handlers in CAPL browser (on diagRequest, on diagResponse, on diagRequestSent) as well as expandable symbolic interpretation of diagnostic messages in trace window will be visible as seen in Figure 20 [35].

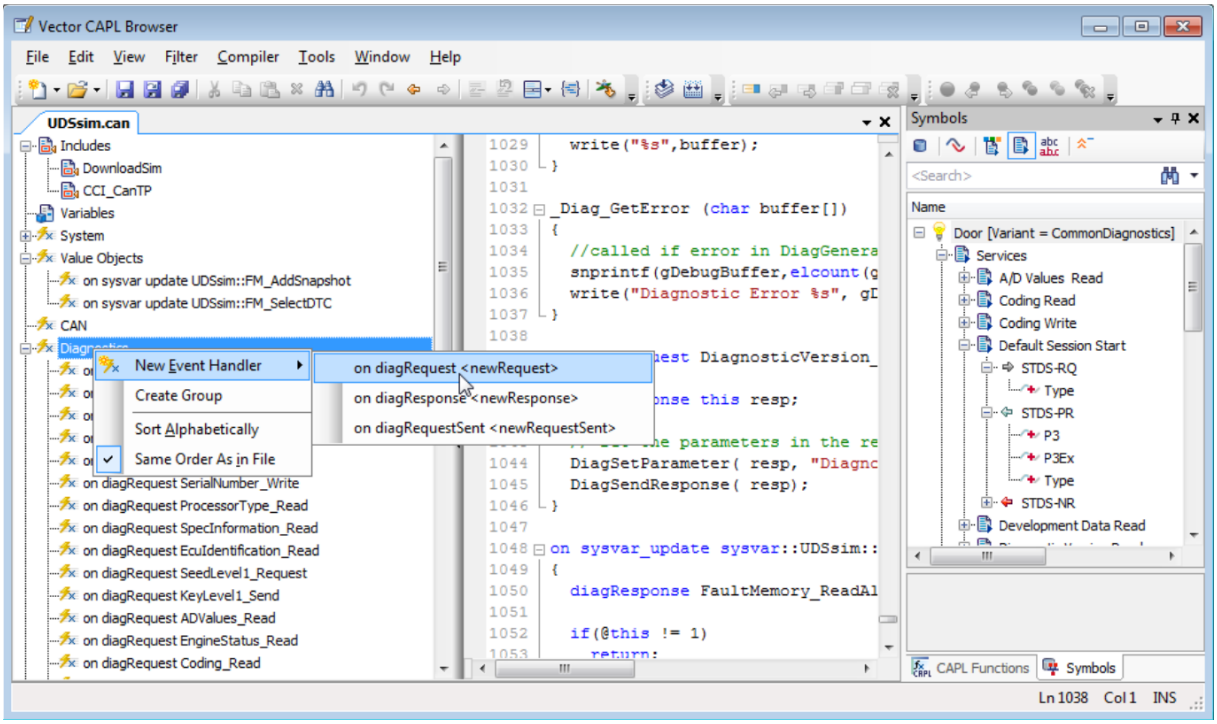

Figure 20 CAPL Browser with diagnostic event handlers [35].

Depending on the diagnostic description either UDS [36] or KWP 2000 will be used as the transport protocol to send and receive requests. For example, to read fault memory (read DTC) service 0x19 in UDS and service 0x18 in KWP will be used. Diagnostic session control (0x10), ECU reset (0x11), clear DTC (0x14), read data by identifier (0x22), write data by identifier (0x2E) are some of the services that can be used to further diagnose a vehicle by using UDS protocol. There are two types of requests that can be defined in a diagnostic description. Physical diagnostic requests are targeted for individual ECUs whereas functional group requests are broadcast in nature and target every ECU found on the network. Request and response objects are handled in CAPL with one object even though that object might exceed the size of a frame capacity. TP module will take care of segmentation and merging of the payload. Figure 21 shows a fault memory list filled with DTCs [37] [35].

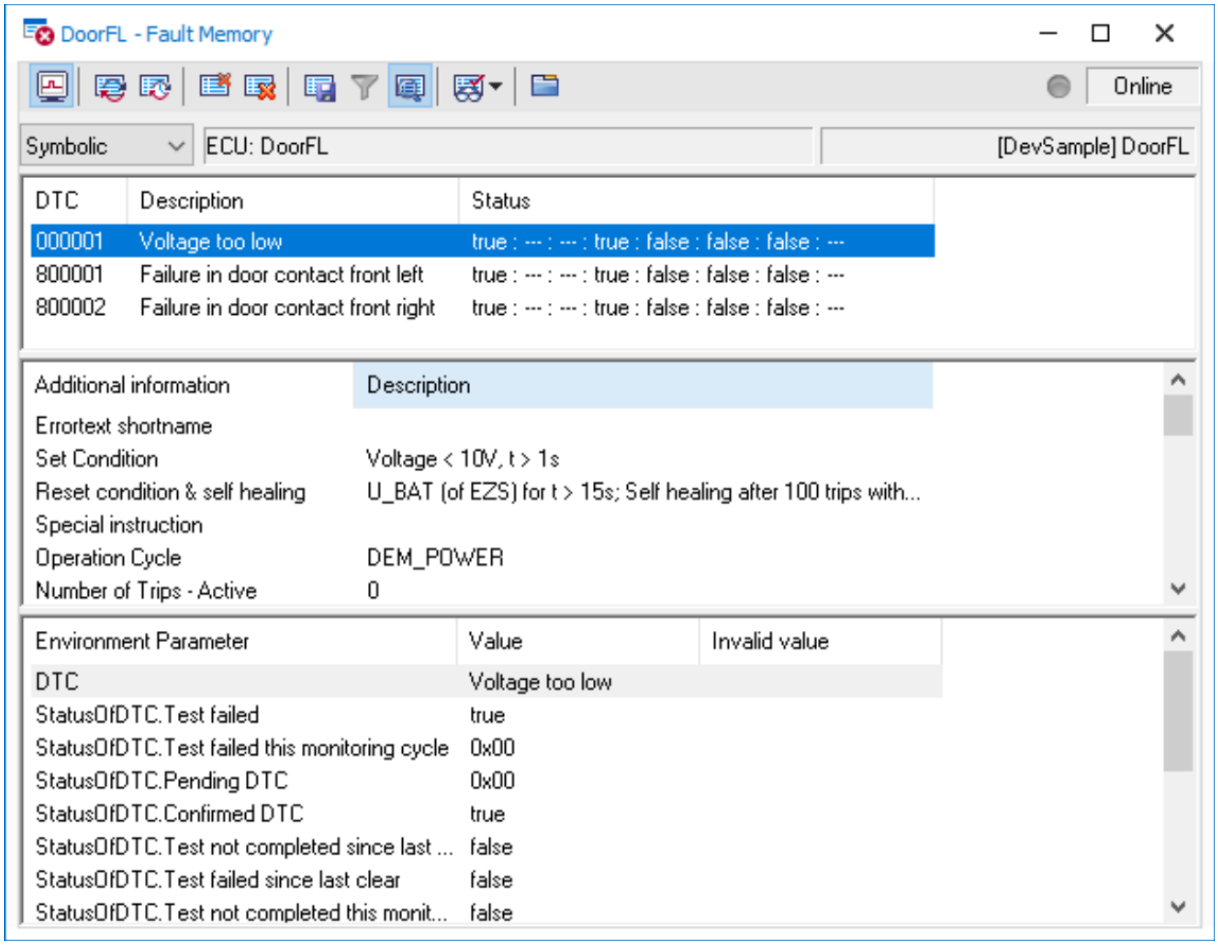

Figure 21 Fault memory window in CANoe [35].

The following CAPL code checks whether a DTC is active in the fault memory list. Function takes only one parameter which is a DTC number. If the DTC is found in the fault memory, then true is returned otherwise false is returned. This function constitutes the barebone of automated test cases targeted to check DTCs.

Table 3 CAPL test script checking a DTC.

```
byte hasDTC(long dtc_check)
{
  DiagRequest FCU101T.FaultMem_ReportDTCByStatusMask req;
  char buffer[100];
  long ret;
  long dtc;
  long count;
  byte i;
  long status;
  byte statusMask = 0x09;  // Read all identified bit field 0000 1001

  // Initialize variables
  i = 0;
  ret = 0;
  dtc = 0;
  count = 0;
  status = 0;

  ret = diagSetParameter(req, "DTCStatusMask", statusMask);
  if (ret != 0)
  {
    return 0;
  }

  ret = diagSendRequest(req);
  if (ret != 0)
  {
    return 0;
  }

  ret = TestWaitForDiagRequestSent(req, 5000);
  if (ret != 1)
  {
    return 0;
  }

  ret = TestWaitForDiagResponse(req, 5000);
  if (ret != 1)
  {
    return 0;
  }

  ret = DiagGetLastResponseCode(req);
  if(ret != -1)
  {
    return 0;
  }

  count = req.GetRespIterationCount("DTCAndStatusRecord");
  for (i = 0; i < count; i++)
  {
    dtc = req.GetComplexRespParameter("DTCAndStatusRecord", i, "DTCRecord");
    req.GetComplexRespParameter("DTCAndStatusRecord", i, "DTCRecord", buffer,
elcount(buffer));
    if (dtc == dtc_check)
    {
```

```
      status = req.GetComplexRespParameter("DTCAndStatusRecord", i,
"StatusOfDTC");
      return 1;
    }
  }

  return 0;
}
```

A sample test case using the above utility function is given in Table 4. The test case first creates an error condition on signal level by setting signal value $\text{FCI\_Stat::FCI\_ErrCat} = 4$ then checks for the existence of DTC 0x1F0028. If DTC is the fault memory, then the test step is a pass. On the second step signal value is set to original value $\text{FCI\_Stat::FCI\_ErrCat} = 0$ and existence of the DTC is checked again. If DTC is cleared, then the test step is a pass again. Informational output statements testStepPass("", "DTC is observed") will be visible on the auto generated test report after all tests in the module completes.

Table 4 CAPL test script for DTC 0x1F0028.

```
testcase TC_DTC_0x1F0028()
{
  @FCI_Stat::FCI_ErrCat = 4;
  testWaitForTimeout(1000);
  if (hasDTC(0x1F0028))
  {
    testStepPass("", "DTC is observed");
  }
  else
  {
    testStepFail("", "DTC must be observed");
  }

  @FCI_Stat::FCI_ErrCat = 0;
  testWaitForTimeout(1000);
  if (!hasDTC(0x1F0028))
  {
    testStepPass("", "DTC isn't observed");
  }
  else
  {
    testStepFail("", "DTC must be absent");
  }

}
```

3 State of the Art

The complexity of the software running on cars increases every day and testing of this software becomes even harder. Systems such as Advanced driver assist systems (ADAS), Adaptive Cruise Control (ACC) and engine controller software need strict timing behavior to function. It is necessary to test the timing behavior of the software on real hardware before installation on a real vehicle. These requirements make HIL testing an indispensable part of software testing for the industry.

Before HIL platforms were common, engineers had to do their testing either on a vehicle or in an integration lab where each ECU was connected to others. Vehicle testing ideally should be done only for calibration and quality assurance purposes. Doing component tests on a vehicle is not a good idea and should be avoided due to multiple reasons. Firstly, they can be complex and very costly. The other reasons are that they can be dangerous, damaging and even impossible. For these kinds of tasks integration labs or HIL simulators should be used.

The first HIL systems were used in the aerospace industry in the 1950s. The first versions were quite expensive around +100K USD and weren't available as off-the-shelf products. Only a handful of companies could afford more than one of them. Without HIL simulators, it was impossible to do standalone ECU tests. In time HIL simulators became more affordable. After the 1990s HILs have found themselves in many industries including automotive [19] [17].

HIL testing can be done in three different levels. These are signal level, power level and mechanical level. Signal level HIL testing only deals only with actual controller board (SUT) and its interface signals. If there are additional power electronics or mechanical loads, then these are simulated. Interface between SUT and the simulator contains only signals.

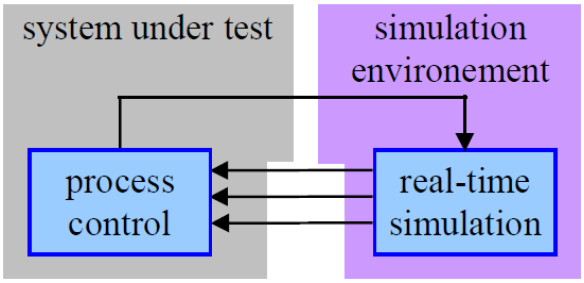

Figure 22 Signal Level HIL simulator [17].

Power level HIL tests power electronics and controller board. In this configuration mechanical loads are simulated. Interface between SUT and the simulator contains signals and power elements. Mechanical level HIL adds an electrical motor to the tested system. Interface between simulator and SUT contains signal, power, and mechanical variables [19] [17].

RCP (Rapid Control Prototyping) reverses roles of plant and controller. The goal of the RCP is to build controllers whereas the goal of HIL to build the plants. RCP tries to develop control algorithms by initial mockups on generic programmable hardware. Generic hardware usually has superior computing power with respect to final hardware. Code is generated based on physical models developed on modeling tools. In RCP, a controller model is developed and uploaded to the generic hardware. Once the model is matured through testing on generic hardware then it can be ported to actual hardware. Whereas in HIL, plant models are developed, uploaded to the target simulation hardware, and used to test actual controller hardware.

For the purpose of this thesis, using a signal level HIL simulator is more than adequate. We can classify HIL setups based on used OS and real-time capabilities. In literature there are many ways to assemble a HIL system. However, for practical purposes these HIL assemblies can be grouped into three categories. These are HIL platforms based on real time operating systems, HIL platforms based in common-off-the- shelf PC and HIL platforms based on common-off-the-shelf PC together with a real time adapter [38]. The following chapters explain each one of them in detail.

In the previous chapters it is shown that HIL systems solved many of the problems that plagued the industry for more than 70 years. HIL platforms are mainly composed of proprietary hardware and software. These platforms are expensive, and their software toolset is complex to master. Companies can’t afford to buy many HIL platforms. Since the number of HIL test beds are limited, developers must wait their turn to test their ECUs. HIL testing platforms have long queues, developers waiting their turn in lines. Due to ever increasing software size and testing frequency, HIL system tests are becoming a bottleneck for software developers. This is especially more prominent in agile methodologies where testing frequency is more than other methodologies. As time spent on testing increases, the cost of the project also increases. Front loading is a solution for this problem. It means bringing tests closer to the bottom of the V-model. The motivation of this thesis is to bring down testing time and testing cost by reducing queues built up in front of HIL stands by introducing front loading. Alternative HIL solutions based on low-cost hardware and software which can be put on desktop will

be investigated. Cost and portability are the requirements of the solution. If we want to put the solution on every developer's desk, then the solution must be cheap and must have a small form factor to fit on a desk. Portability requirements are also enforced by gateway applications. The gateway tests are designed to be done on vehicles. Tester should be able to carry the gateway solution with himself easily. Without a portable HIL platform gateway tests can't be done.

Additional benefits of low-cost solutions are increase in testing frequency, parallelism, and software quality. A proof-of-concept rest-bus and gateway application will be demonstrated. It will be shown that HIL platform based on COTS laptop is a low-cost alternative to established HIL stands. HIL platforms based on PC/laptop have been given different names by different authors. Cost effective HIL and low-cost HIL are some of those names that can be found in the literature.

3.1 RTOS Based HIL Platform

These systems come in various form factors. Although portable versions are also available, they are commonly delivered in rock mounted modules in cabinets. The functionality of these systems can be extended and customized with additional expansion boards. These systems provide different kinds of communication interfaces (CAN, LIN, MOST), hardware-based sensors and actuators (resistors, FPGA, etc.), and a real time OS. Due to their cost, the first HIL systems were used by the aerospace and avionics industries. Later, after cost reductions, those systems became widespread in other industries too [10] [16] [17] [39].

These systems are based on real-time operating systems. That means period fluctuations will be very low compared to non-real-time operating systems. In other words, these systems are hard real-time capable. Depending on the toleration of the tested system, hard real-time capability might be a requirement.

Figure 23 Signal flow in HIL simulation [16].

Not only software functional tests but also non-functional tests such as electrical tests can be done on RTOS based platforms. Electrical fault injections tests include short circuit tests and open circuit tests. Due to its real-time-based operating system it is best suited to hard real time required environments. However, it is not the only tool that is hard real-time capable. Figure 23 shows a HIL simulator from dSpace. Their direct integration MATLAB/Simulink toolboxes make them an ideal choice for industries employing MBD (Model based development). They come in many sizes. Smaller ones also exist to fit on a desktop, but certain functionalities need to be stripped down [16] [38].

The disadvantages of HIL platforms based on RTOS are their proprietary complex hardware and software toolchains. Complexity of the toolchain is accompanied by a steep learning curve. Compared to the off-the-shelf solutions, these dedicated HIL platforms are more expensive. Most companies can afford only a small number of these systems. Because of that developers have to wait in long queues to test their ECUs. Although they provide higher testing frequency compared to on-vehicle tests, off-the-shelf solutions provide better testing frequency by increasing parallelism [38].

3.2 COTS Based HIL Platform

Figure 24 Cost effective HIL platform based on COTS software and hardware.

Figure 24 represents a HIL platform based on common-off-the-shelf hardware; a laptop is connected to the SUT (System under test) / DUT (Device under test). Communication interface hardware is either a USB CAN box or a PC card available off-the-shelf.

Most notable advantages of this system are being relatively cheap and extremely portable. Compared to RTOS based HIL benches the cost of a COTS laptop and software is just a fraction of it. Every developer in a company is already given a laptop. All the equipment used on this platform is usually available in the office. That means the actual cost of bringing together this system is very low. The COTS software we are going to use is the Vector CANoe tool. It is a Swiss army knife used in the automotive industry not only used for analysis and diagnosis but also used for simulation and testing purposes as well. Operating system (OS) running on a laptop is a non-real-time based OS, typically Windows OS. Using a non-real-time OS is preferred when integrating HIL platform to the existing tools used in model-in-the-loop (MiL) and software-in-the-loop (SiL) are required. CANoe tool is used to create virtual simulated nodes to do the rest-bus simulation. For the rest-bus simulation of the FCU, a CAN interface with 4-ports is necessary. Each port will be used to connect corresponding ports of the FCU. Gateway application on the other hand requires only two channels. Gateway intercepts messages flowing in a CAN bus. The first channel will be used for collecting incoming messages and channel 2 will be used for delivering messages from the other end. It will function like a pipe to relay messages coming from one port and leaving from another port. ECUs have lots of IO pins. CAN pins are the only pins we can interact with this toolset. Usually, ECUs use those IO pins to have a connection to sensors and actuators. These types of sensors don’t directly communicate through the CAN interface. The downside of this setup is that it needs additional hardware to simulate those sensors. One solution is to use data acquisition (DAQ) hardware. This

type of hardware and a PC can be used to simulate external sensors. However, due to time limitations and additional costs sensor simulations connected to IO ports won't be covered by this thesis.

As long as ECU is tolerating period deviations of messages coming from rest-bus simulation, it will be shown that such systems are viable alternatives to established high-end HIL benches even though their real time behavior is not on par with them. Source of deviations are OS task scheduler, other processes running in the background such as antivirus software, time lost during arbitration, message transfer delay caused by both CAN and USB connection. Software related deviations attributed to the OS scheduler and other processes competing for CPU time. Hardware related delays attributed to arbitration time lost to other higher priority messages and actual transfer time of messages in CAN and USB connection. Among these delays, software-related ones constitute the major slice of the total lost time. Speed of USB protocol is also another factor contributing to the latency. Detailed latency and periodicity experiments done in the following chapters will prove these statements [10] [38].

3.3 COTS and RTA Based HIL Platform

RTA (Real time adapter) is an extension hardware added in between SUT and the laptop to increase real time behavior of signals [38]. Some authors prefer to name this extension an IHIL (Interface HIL) [10]. This extension can be an MCU (Microcontroller unit), FPGA (Field Programmable Gate Array) or any custom hardware. Basic idea behind adding an RTA is to increase real time behavior of signals coming from rest-bus simulation in a laptop. This is achieved by offloading some functionality from simulation running in a laptop to the interface hardware. CANoe has the ability to split its real time core from the main analysis part. This allows real-time dependent parts to run on external hardware while keeping analysis parts run on the host computer. Some CAN interface products from Vector in addition to being a CAN interface, allow executing CAPL code on them too. This mode of operation, in other words moving a part of the program from the computer to the hardware is called CAPL-on-board. The CAN interface box used in our experiment is a low end one and because of that some CAPL libraries are nonfunctional. Having said that, we will demonstrate CAPL-on-board in one of the periodicity experiments. CAPL-on-board mode is equivalent to adding an RTA extension done in literature. Both solutions bring down periodicity violations, standard deviation, and outliers to considerable degrees. The downside of this solution compared to the previous one is the added RTA cost. Having said that, this platform is favored by academia especially implemented with open-source

software on cheaper development boards to do research or teach the subject in the labs.

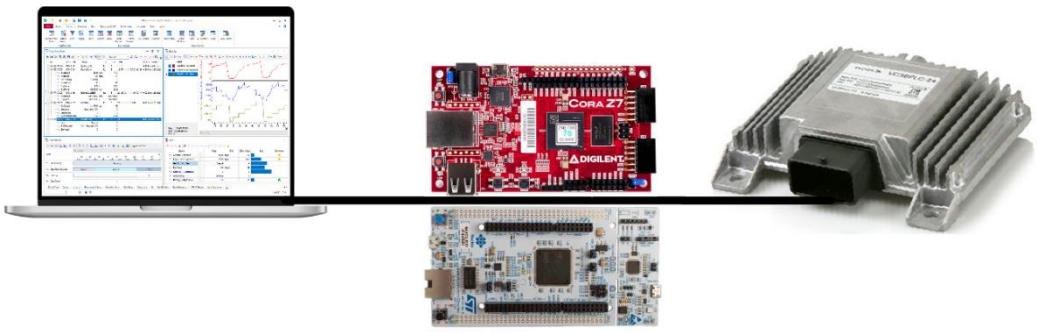

Figure 25 Cost effective HIL platform based on COTS and RTA.

To summarize, each solution has its advantages and disadvantages. Table 5 is prepared based on information found in literature and can be used to cross-compare each solution.

Table 5 Comparison table of state-of-the-art HIL platforms.

HIL Platform	Testing Time	Testing Cost	Platform Cost	Portability	Testing Frequency	Parallelism	Real-time	Complexity
RTOS based HIL	High	High	High	Low	Low	Low	Hard	High
COTS based HIL	Low	Low	Low	High	High	Yes	Soft	Low
COTS and RTA based HIL	Low	Low	Low	Med.	High	Yes	Hard	Med.

RTOS based platforms are the most expensive ones. One reason for this is because it has more features. Functional tests as well as non-functional tests such as open circuit and short to ground type of electrical fault tests can be carried out. Besides, the computers used in those systems are very high-end and contain more processing power. Modular architecture allows them to scale themselves to the needed sizes. Due to having proprietary hardware and software toolchains, RTOS systems are complex and have a steep learning curve. Low-cost COTS solutions come with common desktop hardware and software found on the market. Those HIL solutions can be built

with a laptop only. Besides being the cheapest, they are simpler and most portable ones. Since they can be deployed on every developer's desk, they increase parallelism and testing frequency. However, their real-time capabilities are not at the same level as RTOS solutions. Hard real-time systems generate more accurate signals compared to the soft ones. As long as timing requirements are satisfied then those solutions are worth exploring. RTA based solutions fill in the midpoint between COTS and RTOS solutions and deliver a good price-performance ratio. Their processing power is at the same level as COTS solutions, but their real time capabilities are on par or in some cases might be even better than RTOS solutions due to their hardware-based extension. Overall, the last two solutions have better testing frequency. They have small form factors and can be stationed on each developer's desktop. Portability is another requirement that must be considered when choosing one platform over the other. RTOS HIL cabinets are the least portable ones while COTS laptop solutions are the most portable ones. Portability allows them to be carried even in a vehicle. The closer the testing platform is to the developer the better. The concurrent access to the testing platform allows developers to do tests as much as they want. Increased parallelism increases testing frequency. This also makes the last two solutions favored in agile software development methodologies. Agile methodologies are iterative in nature and do a lot of integration and acceptance tests due to their involvement with the customer in the early stages of the development. Contrary to the agile methodologies, waterfall methodology on the other hand does all integration and acceptance tests at the end of the project [38].

Testing is logically accompanied by automation. Testing can be done either manual or automated fashion. All solutions presented above have the capability to do automated tests. Automated testing has additional benefits, and those benefits are explored in Chapter 2.2.

Each solution has its benefits, and it is hard to select the best one. It is more logical to choose a solution based on a use case. The COTS based HIL solution was chosen because it satisfies all motivational requirements. Those requirements were to reduce testing time and testing cost. In addition, this solution is feasible because it is cheap and yet the most portable one. Having said that, our aim is not to replace RTOS based HIL platforms with low-cost ones. RTOS HILs will keep their place as it is because not every test can be done on low-cost platforms. The aim here is to add another HIL testing system closer to developers and let them clear their faults before the actual HIL testing phase.

4 Methodology

To demonstrate a proof-of-concept solution on a low-cost COTS HIL platform on desktop it is important to do some groundwork to prove its feasibility first. It is worthwhile to explore latency and periodicity of the signals in a simulated environment. A HIL simulator blends simulated virtual nodes with real hardware nodes. Conceptually simulation speed must cope with speed of the real hardware. Otherwise, the whole idea of merging real hardware with simulation wouldn't be feasible. The first two subsections of this chapter will investigate latency and periodicity of messages on a running simulation. Objective of these subsections is to determine limitations of the proposed HIL solution.

CAN communication is signal oriented. Signals are carried by messages in the payload section of a CAN frame. To place a signal in the payload, its starting index and length must be known beforehand. Signal definitions are either done in a database (e.g. CANdb++) or programmatically. Each node participating in a CAN bus transmits its messages by broadcasting it to the remaining nodes. Once a message is put into the CAN bus, due to being a shared broadcast medium, all other nodes receive it. However, transmission of the message and reception of the message will be at different time points due to the propagation latency of the message. It is enlightening to investigate message propagation time on the CAN bus especially in a half simulated half real environment. In the following chapters, signal propagation time will be calculated theoretically and will be compared to the actual measured values. Then the same experiment will be repeated for period measurement of cyclic messages. The periodicity experiment measures the period between two successive messages received by the receiver node. Depending on the application latency can play a significant role. It is desirable to have a very low latency in ACC and ABS systems. If a velocity message originating from a wheel speed sensor arrives late either due to propagation latency or a defect in periodicity of the message, then these systems won't respond for a short period of time. Small interruptions in the functionality of those systems will result in damage to the car or endanger the passenger. In literature these kinds of experiments are done with different hardware and software tools. One of the goals of following subchapters is to redo the same experiments done earlier in literature and compare results with them.

Rest-bus simulation is software technology that allows replicating missing ECUs in a network with their virtual counterparts. Without rest-bus simulation, HIL tests of a single ECU can't be done in a standalone environment. The usual workflow of simulation is

to do them in three phases. In the first phase, the remaining bus simulation is done for all nodes in the bus. All nodes are simulated without exception. A prerequisite for this operation is having a CAN communication matrix database attached to the bus. The database file contains the communication matrix of the entire network. Once the database is there, the remaining work is to create virtual nodes and program them using CAPL language. A test run done in a fully simulated environment will give insight into the feasibility of the whole system. If the communication among nodes is successful and bus load is within allowed ranges, then the first step of workflow is assumed to be successful. In the time of development phase real hardware might not be available. These virtual nodes will serve as prototypes (mockups) of real hardware until real hardware becomes available. CANoe in this scenario serves as an RCP (rapid control prototyping) tool. This type of workflow also encourages RCP. Replacing one virtual node with its real counterpart one at a time is the foundation of RCP (rapid control prototyping). RCP is a development methodology focusing on developing controllers themself and can be regarded as complementary to HIL which focuses on plant simulation. The second step of the workflow is to replace one virtual node with its real counterpart. This mode is what is being done on HIL stands. A real ECU is connected to a CAN bus and all other participants in other words the remaining bus is simulated. Virtual nodes developed for rest-bus will serve as a basis when modeling real ECU functionalities. Each real ECU is tested standalone in this way until all ECUs are made available. The last step of workflow is to replace all virtual nodes with their real counterparts. This mode of operation is nothing but an integration test. When all nodes are replaced with their real counterpart the whole test becomes an integration test. CANoe will serve as an analysis and testing tool. Simulation part won't be used anymore. Once integration tests are done, ECUs are ready to be installed on vehicles.

4.1 Latency

Each SUT has different timing requirements and tolerance. For some systems deviations in periods and increase in latency are tolerable and violations won't pose any issues. Having said that, for other systems signal's period and latency deviations could create dangerous situations. For example, an engine ECU needs to respond timely to sensor signals. Engines have fast-moving parts and a small one millisecond delay in sensor signal might cause either a failure or a damaging situation to the running engine. Suppose ECU cannot respond to sensory values in time. The timing of the engine will be broken and will result in a non-functional engine or even a damaged one. Same is true for ADAS systems. Those ECUs also need to respond to sensory values in a timely manner. One millisecond delay may result in the Anti-lock

Brake System stopping the car a few milliseconds later. At high speeds a few milliseconds determine whether the car will be involved in an accident or not [39].

Knowing latency times beforehand is valuable when assigning nodes to a bus in the design stage. Theoretically CAN bus can accommodate unlimited nodes; however, latencies will increase, and performance will suffer in overloaded buses.

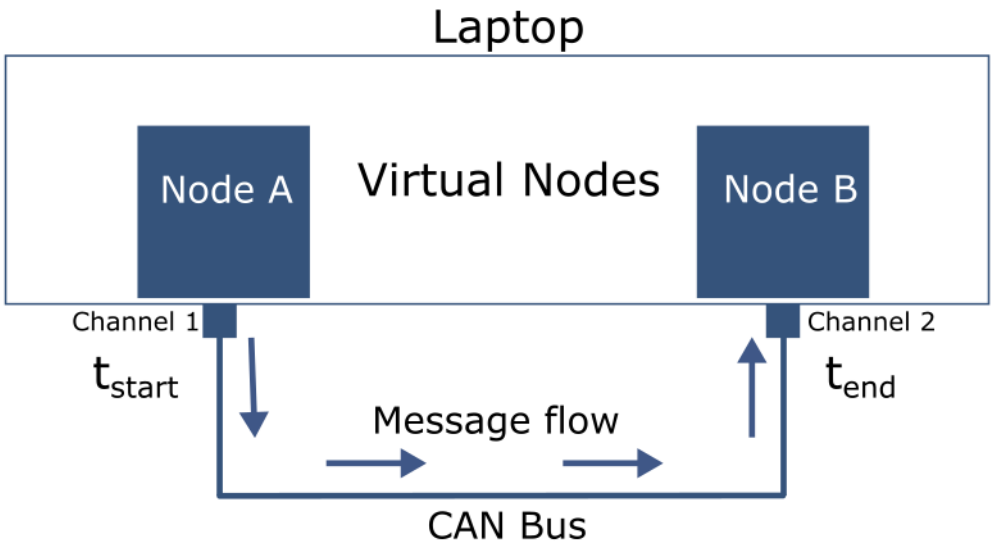

Figure 26 Conceptual representation of the Latency Experiment.

In Chapter 2.4 latency induced by CAN bus itself in the absence of other nodes are theoretically calculated. Frame transmission times both for FD and classic frames are calculated. For FD extended frames the result was around 0.24ms, while for classic extended frames the result was around 0.31ms. Both results were very close to each other. It is safe to assume that latency caused by CAN bus itself in the absence of other nodes is one third of a millisecond. The question is how much delay the other variables contribute to the latency? Other variables are nodes themselves and the USB line between laptop and CAN interface hardware. Nodes run on a non-real-time OS. Windows OS has many processes running in background (antivirus software, mouse events, keyboard events, paint events etc.) which will negatively impact performance of the scheduler. The USB line connecting the laptop to the CAN interface will also contribute an additional latency to the system. It is hard to calculate all those random contributions on the paper. It is easier to devise an experiment and measure total signal latency experimentally. Figure 26 depicts a concept experiment devised to investigate how much signal latency is present in the proposed solution. Here node A is the sender node and node B is the receiver node. A periodic message is generated in node A and the current time of node A is put into its payload.

Table 6 CAPL script Node A in latency test.

```
variables
{
  mstimer t_10ms;
  message 0x100 m_100 = {dlc = 8};
}


on start
{
   setTimer(t_10ms, 10);
}


on timer t_10ms
{
    setTimer(t_10ms, 10);
    m_100.double(0) = timeNowNS();
    output(m_100);
}
```

Node B receives transmitted messages and extracts the payload data containing time of node A. Then node B subtracts the current simulation time of node B from the time which was delivered in the payload. The difference between the end time and the start time is defined as latency.

$$Latency = \Delta t = t_{end} - t_{start} \quad (6)$$

Payload data is extracted from the message in event block and $this.double(0)$ contains the first double value in the payload. Table 7 shows a portion of a script belonging to node B.

Table 7 CAPL script of Node B in latency test.

```
variables
{
  const double nano2ms = 1.0e-6;
}

on message CAN2.0x100
{
  if (this.dir == RX)
  {
    @Gateway::latency = (timeNowNS() - this.double(0)) * nano2ms;
  }
}
```

Latency is measured for a periodic message for a duration of approximately 60 seconds. Latency vs time graph is given in Figure 27. Spikes represent deviations, and outliers. Figure shows that latency fluctuates around some average value.

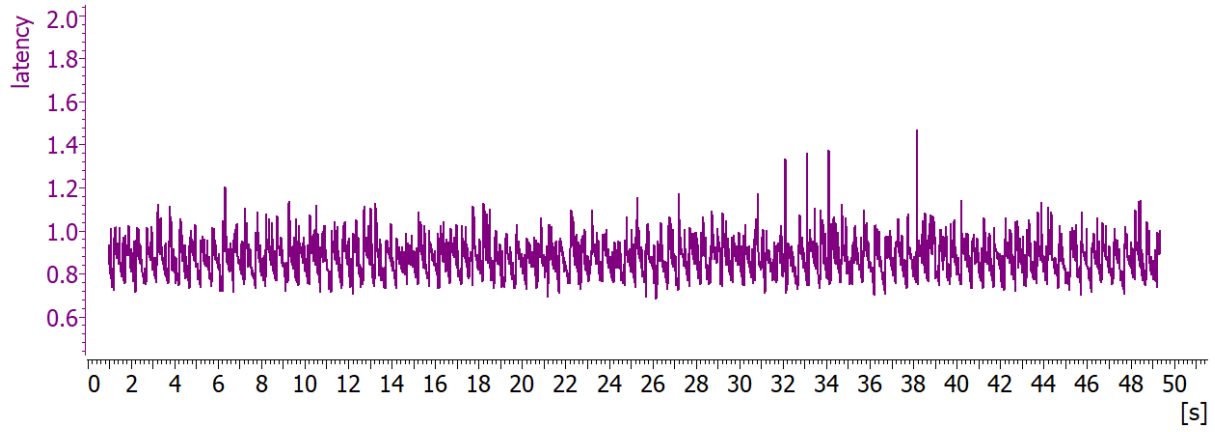

Figure 27 Latency vs Time graph of messages, cycle time is 10ms.

The observables here are min period, max period, average period, and standard deviation. Repeated tests showed us that the mean value is floating in some range [0.6ms, 1.5ms] in every repetition of the test. In other words, measured average value is not consistent and changes in some range from test to test. Figure 28 shows the same measurement but this time average (mean) value is different than before.

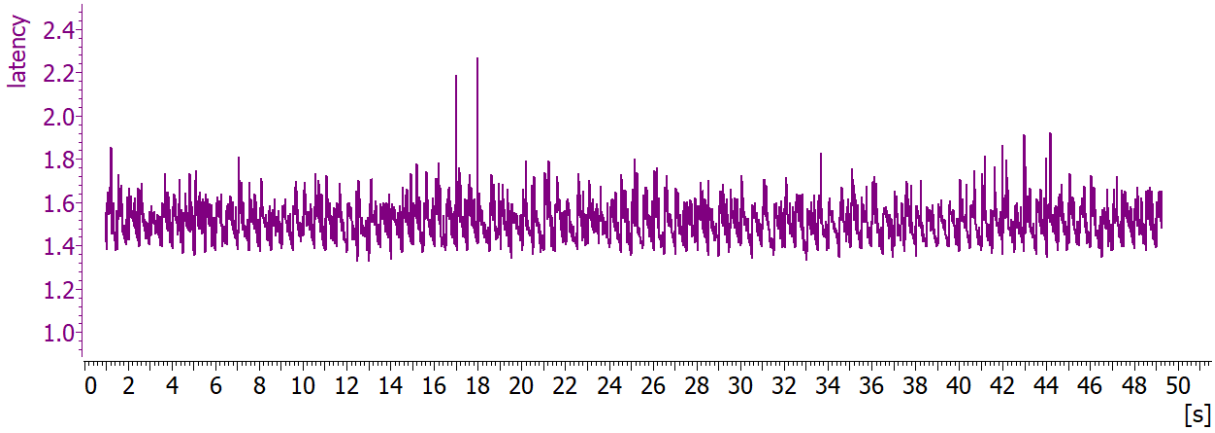

Figure 28 Latency vs Time graph of messages, cycle time is 10ms.

It is suspected that this discrepancy is the result of other processes such as background processes, antivirus software, keyboard events, and mouse events competing for CPU time that happens to be the characteristics of a non-real-time OS. Scheduler that comes with a non-real-time OS doesn't have the most reliable timing behavior compared to the hard real-time capable systems.

Table 8 Latency tests, bus is empty, cycle time is 10ms.

Name	Period	Min	Max	Average	Standard Deviation
Latency	10ms	1.3292	2.2672	1.5123	0.0805
Latency	10ms	1.2937	2.1077	1.4847	0.0838
Latency	10ms	1.1872	3.4152	1.3833	0.1445

Latency	10ms	0.6800	3.4300	0.9175	0.1504
Latency	10ms	0.5845	2.5965	0.8800	0.1457
Latency	10ms	0.6829	1.4709	0.8750	0.0812
Latency	10ms	0.6621	2.7081	0.8763	0.1321
Latency	10ms	0.4312	0.9892	0.6199	0.0787

In real world contention due to arbitration of the CAN messages resulting from shared medium, will also contribute to the delay a bit more. Table 8 lists all measurements done for a signal having the period of 10ms. Standard deviation as well as min, max outliers depend on capabilities of the platform. Same test repeated for different periodic messages. The tests are repeated on an empty bus for 1ms, 10ms, 50ms and 100ms periods. Independent from the chosen period, measured average value still slides in [0.6ms, 1.5ms] range. Based on measured data, we can conclude that the chosen period doesn't have a measurable impact on the latency average value slides.

Table 9 Latency tests for different cycle times, bus is empty.

Name	Period	Min	Max	Average	Standard Deviation
Latency	1ms	0.3226	1.6526	0.8487	0.2377
Latency	10ms	1.2937	2.1077	1.4847	0.0838
Latency	50ms	0.6798	1.2158	0.8730	0.0851
Latency	100ms	0.5733	1.0130	0.7616	0.0841

It is desirable to observe testing platforms behavior under different stress levels. To see the effects of the congestion to the latency, the experiment is modified to include additional nodes to simulate a load. Artificial load is generated by an interactive generator (IG) of CANoe as seen in Figure 29. Table 10 and Table 11 show that averages, deviations, and outliers are getting worse when experiment is conducted under the load. These results are aligned to our expectations. The more load on the testing environment the more delays expected to happen.

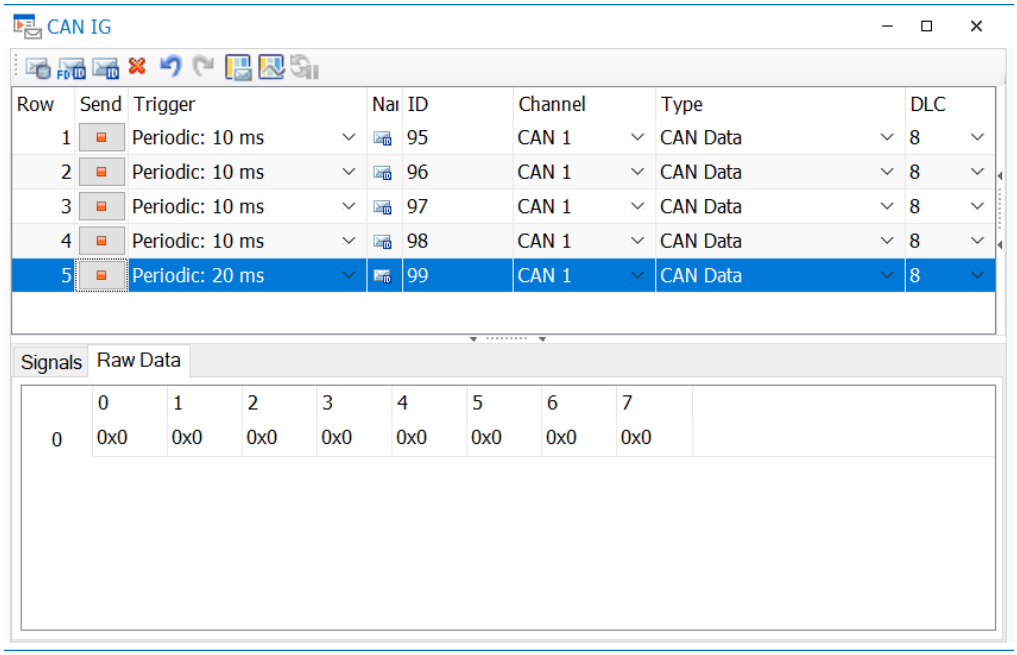

Figure 29 Artificial load is created by a generator.

Delay factors causing the latency are OS scheduler both from sender and receiver side and the contention happening on the bus. If the CAN bus is empty, we will not observe contention effects. The only delay happening on CAN bus originates from message transmission time and that was calculated to be around 0.3 milliseconds.

Table 10 Latency tests for different cycle times under artificial load.

Name	Period	Min	Max	Average	Standard Deviation
Latency	1ms	0.3051	2.1711	1.0991	0.3519
Latency	10ms	0.4630	2.4416	1.1510	0.4026
Latency	50ms	1.6677	2.6497	2.0693	0.1626
Latency	100ms	1.8400	2.4040	2.0080	0.0967

Repeated tests for a single period under load, here 10ms, shows us the average value is sliding in some [1.1174, 1.7714] range as happened with empty buses. When Table 8 and Table 11 compared it is seen that fluctuations of min, max and average values became worse in later cases.

Table 11 Latency test for 10ms periodic message under artificial load.

Name	Period	Min	Max	Average	Standard Deviation
Latency	10ms	0.6312	2.4252	1.3143	0.3990
Latency	10ms	1.1002	2.7902	1.7714	0.4003

Latency	10ms	0.4190	2.1710	1.1174	0.3997
Latency	10ms	0.8753	2.5633	1.5518	0.3999

4.2 Periodicity

Periodicity experiment has the same layout as in the previous latency experiment. Figure 30 represents the overall layout. Node 1 is the sender node whereas node 2 is the receiver node. An artificial load node generator is attached to the sender part. The generator node creates some arbitrary messages to put stress on the bus and OS scheduler. Experiment is done with and without an artificial load and then results are compared.

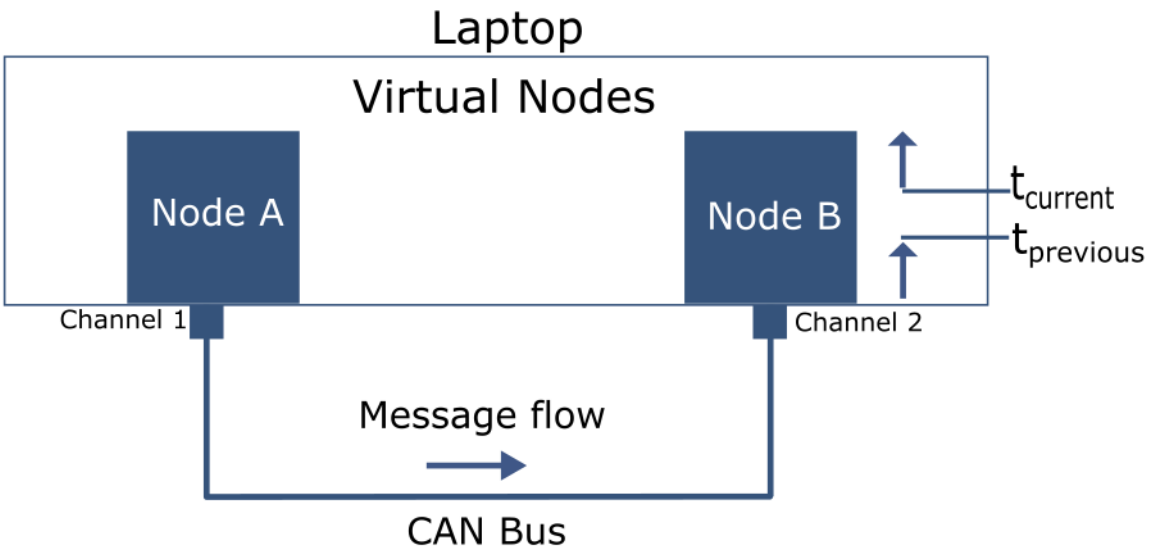

Figure 30 Simulation Setup of periodicity experiment.

Contrary to the previous experiment, this experiment measures the time between two successive messages arriving on the receiver side. Corresponding CAPL code is given Table 12. On the top two variables are defined to hold a timer and a message. The timer triggers every 10ms. At each timer interrupt program sends the message to the CAN bus. Contrary to previous latency tests, we don't need to put the current time into the payload. Messages are sent without a payload.

Table 12 CAPL script of Node 1 in periodicity tests.

```
variables
{
  mstimer t_10ms;
  message 0x100 m_100 = {dlc = 8};
}

on start
{
```

```
   setTimer(t_10ms, 10);
}

on timer t_10ms
{
    setTimer(t_10ms, 10);
    output(m_100);
}
```

CAPL code corresponding to the receiver node is given in Table 13. To measure the time difference between two successive messages, it is necessary to define two variables. These are used for storing the time point of the previous message and time point of the current message. After receiving each message, the program calculates the time difference between two successive messages. This difference in Equation (7) is defined as the period.

$$Period = \Delta T = t_{current} - t_{previous} \quad (7)$$

Table 13 CAPL script of Node 2 in periodicity tests.

```
variables
{
  const double nano2ms =   1.0e-6;
  double message_previous_time = 0;
  double message_current_time = 0;
}

on message CAN2.0x100
{
  if (this.dir == RX)
  {
    if (message_previous_time != 0)
    {
      //message_current_time = timeNowNS();
      /* CAPL-on-board */
      message_current_time = this.time_ns;

      @Gateway::message_period = (message_current_time - message_previous_time)
      * nano2ms;
      message_previous_time = message_current_time;
    }
    else
    {
      //message_previous_time = timeNowNS();
      /* CAPL-on-board */
      message_previous_time = this.time_ns;
    }
  }
}
```

Due to the non-deterministic nature of the non-real-time OS, measured period is observed fluctuating around an average value. This average value is very close to the

period itself. Causes of fluctuations include using a non-real-time OS and CAN bus itself. When the bus is empty, fluctuations originating from arbitration are non-existent. On the contrary if there are other messages flowing in the bus, this time CAN bus also contributes to the fluctuations. This is due to losing arbitration to higher priority messages.

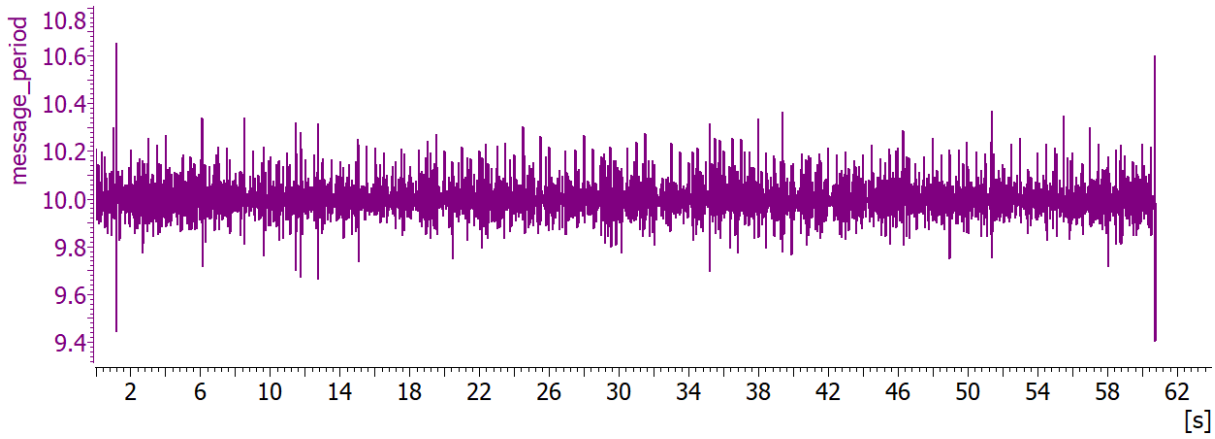

Figure 31 Periodicity vs Time graph of messages, cycle time is 10ms.

One important observation here is that unlike latency experiments, average values observed in this experiment are quite consistent and very close to the actual period of the messages. Whatever caused the average value fluctuations in the latency experiment, in period measurements those effects are canceled out when we subtracted two messages and resulted in a steadier repeatable measurement. Figure 31 and Figure 32 shows periodicity experiments for two periods on an empty CAN bus.

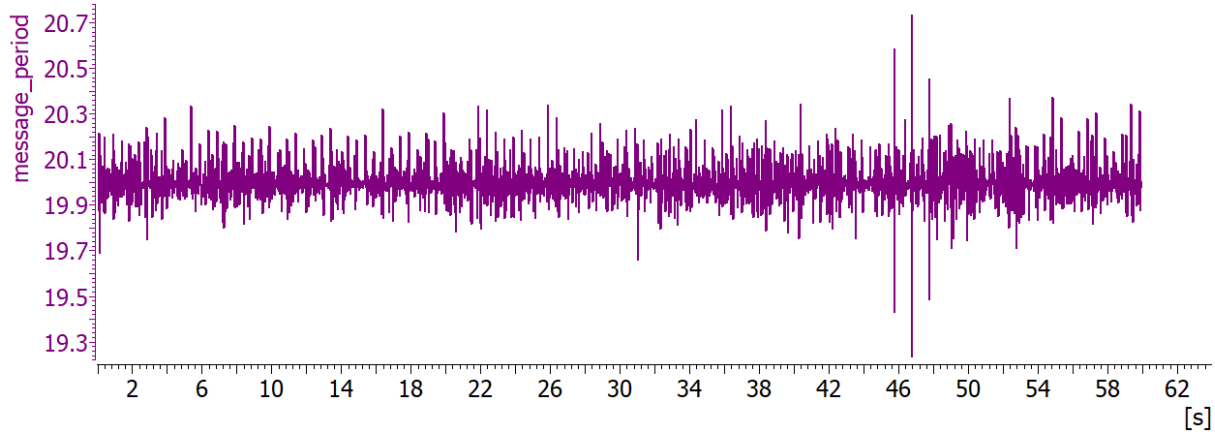

Figure 32 Periodicity graph of 20ms messages under no load.

Experiments are repeated for different periods and each result is recorded in Table 14. While in periods bigger than 1ms, proportionally speaking outliers' deviation compared to the average period is negligible, they are very striking for 1ms messages. Simulation platform barely copes with messages of 1ms. This result is aligned to latency

experiments done earlier. Fluctuations causing latency are also causing fluctuations in outliers in periodicity.

Table 14 Period vs Time graph for cyclic messages under no load.

Name	Period	Min	Max	Average	Standard Deviation
Periodicity	1ms	0.2520	1.7280	1.0000	0.0714
Periodicity	1ms	0.2520	1.7460	1.0000	0.0408
Periodicity	10ms	9.3480	10.6840	10.0000	0.0759
Periodicity	10ms	8.4380	11.5640	10.0000	0.2143
Periodicity	10ms	9.4060	10.6540	10.0000	0.0654
Periodicity	20ms	19.6220	20.4240	19.9999	0.0875
Periodicity	20ms	17.2680	22.6600	20.0000	0.2001
Periodicity	20ms	19.2360	20.7340	20.0000	0.0829
Periodicity	20ms	19.5500	20.3860	20.0000	0.0785
Periodicity	50ms	49.6740	50.4920	49.9999	0.1045
Periodicity	100ms	98.9220	100.8940	100.0000	0.1760

If the bus is busy, delays caused by the bus itself (losing arbitration) also contribute to the period fluctuations. Standard deviation and min, max values also increase. Figure 33 represents periodicity fluctuations for a message whose cycle time is 20ms under a load.

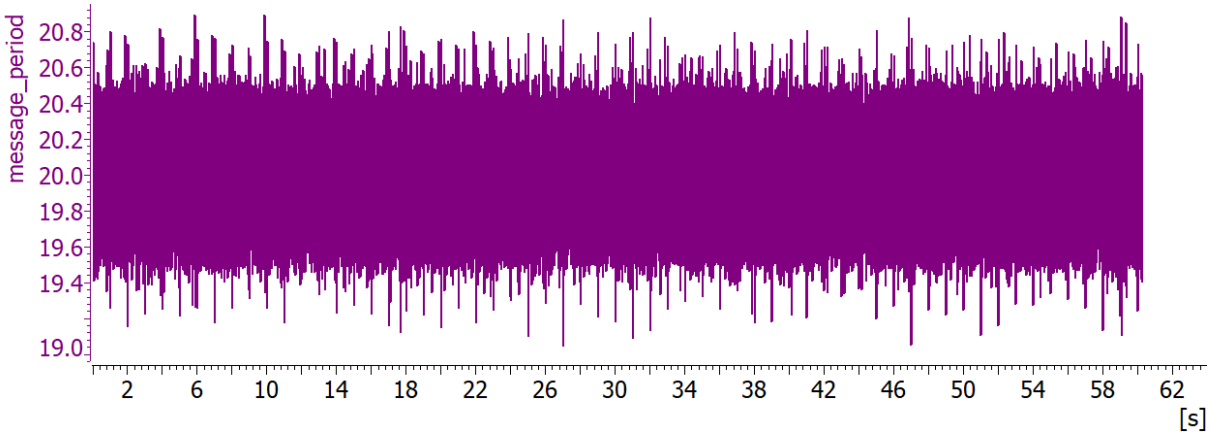

Figure 33 Periodicity graph of 20ms messages under load.

Table 15 exhibits periodicity of measurements for four different periods under artificially generated load. When Table 14 and Table 15 compared, the most notable observed differences are increase in standard deviations as well as the min-max outliers, but average values remain the same. For the case of 20ms period messages, max values reach up to 3ms. Depending on the application, this kind of deviation might pose a risky situation to the controller. Some controllers might even generate a DTC if

measured periods deviate too much from the expected value. As long as the controller tolerates and doesn't generate a DTC, it is assumed that deviations are being tolerated by the controller. Keep in mind that artificially generated load is relatively small (10%) compared to rest-bus simulation that we will do in next chapters. We expect deviations to get bigger when the load gets bigger.

Table 15 Periodicity vs Time under load.

Name	Period	Min	Max	Average	Standard Deviation
Periodicity	1ms	0.2520	2.6780	0.9994	0.6172
Periodicity	10ms	8.5140	11.4000	10.0000	0.6786
Periodicity	10ms	6.8900	12.9540	9.9999	0.7229
Periodicity	20ms	18.8600	21.1840	20.0001	0.5282
Periodicity	20ms	16.6740	23.2440	20.0001	0.5999
Periodicity	20ms	19.0440	20.8960	20.0000	0.5102
Periodicity	50ms	49.2420	50.7020	49.9999	0.2637
Periodicity	100ms	99.5980	100.5060	99.9996	0.1459

The most negatively affected period is again 1ms message. Max value is 2.6 times higher than the period itself. This experiment proves that deviations up to 3ms are expected for all periods at %10 bus loads. If the load gets bigger deviations are expected to get bigger too. It is safe to say that simulating messages for systems that have little tolerance to the deviations for periods around 1ms is not advised within our proposed solution. Even 5ms messages might not be feasible under heavy loads due to increased deviations. The result obtained in this experiment recommends us to use the 10ms period as the lowest period to simulate low toleration systems. Another conclusion that can be drawn from the above table is that no matter which period is used, the testing platform produces impeccable average values.

This kind of experiment is done earlier in literature on a different hardware and software setup and results can be seen in Figure 34. Although experiments are done on different hardware and software, results are similar to ours. Statistically speaking results obtained in our simulation environment have better standard deviation, min, and max values. Figure 34 has a standard deviation of value 0.878 whereas our tests have 0.5282 as seen in Figure 32. Likewise, Figure 34 has outliers at min 18ms and max 26ms whereas our tests have them at min 18.86ms and max 21.18ms which again have better results.

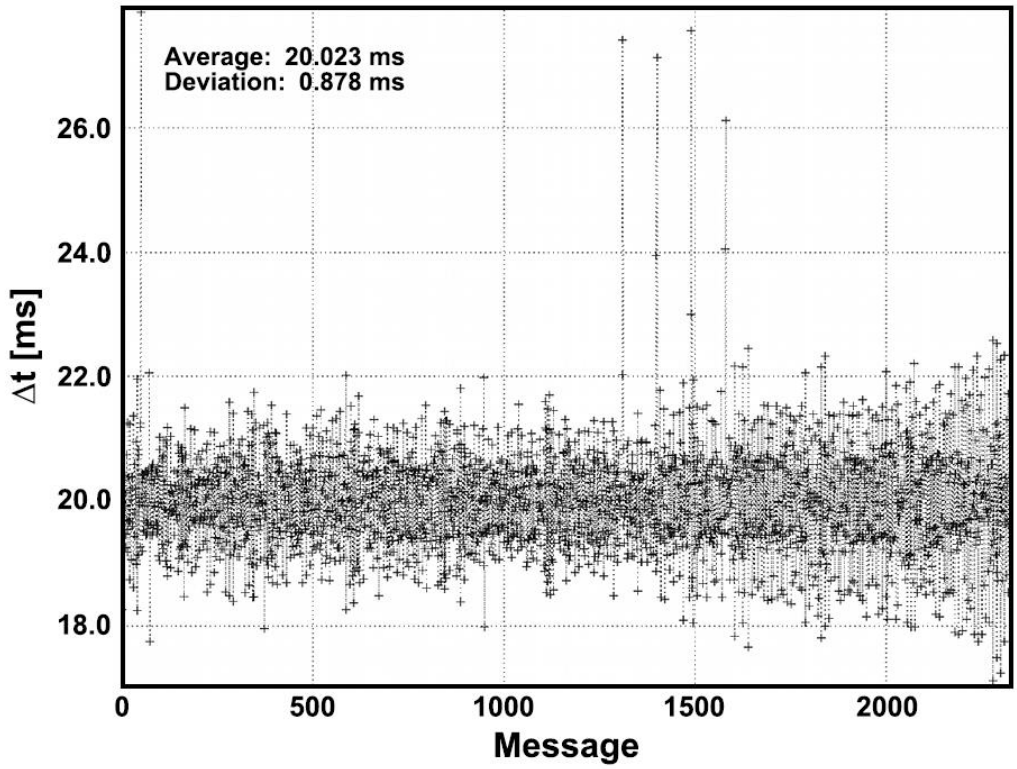

Figure 34 Non RTOS HIL shows period fluctuations for a message cycle time is 20ms [38].

Hardware and software tools used for testing constitute the testing environment. Hard real-time capable environments such as hardware-based equipment create smaller deviations and outliers, whereas software based soft real-time capable environments generate bigger deviations and outliers. CANoe has the ability to move CAPL scripts of virtual nodes to the hardware. Figure 35 shows the CAPL-on-board configuration window of a node. Scripts put on the simulation window can be moved but scripts put on the measurement window can’t be moved on hardware.

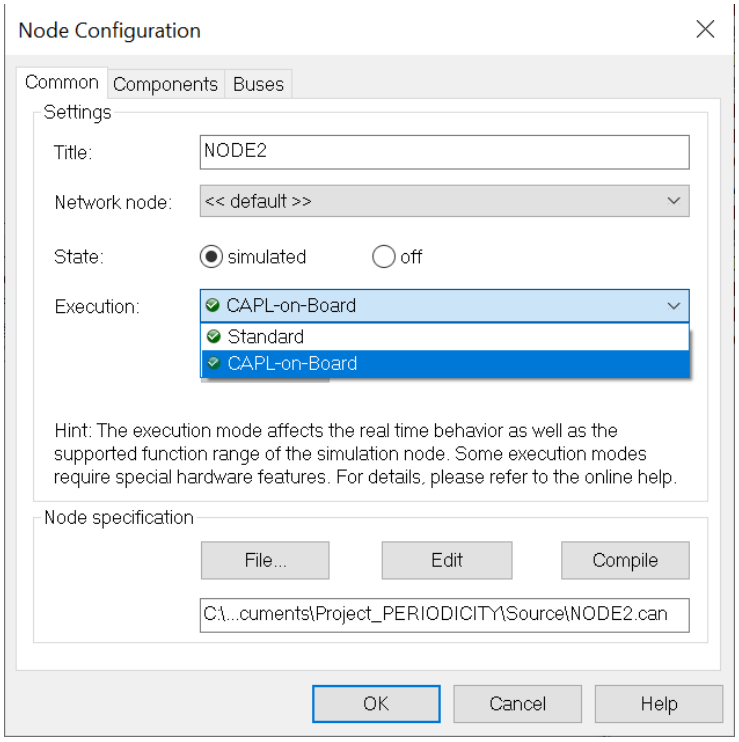

Figure 35 CAPL-on-board configuration of a node.

For our purposes this limitation doesn't pose any problem. The ability to move scripts to the hardware is called CAPL-on-board mode. In some literature this mode is named RTA (real time adapter) and even HIL interface mode. The difference between CAPL-on-board mode test results and the regular software-based test results are huge. Deviations in CAPL-on-board mode are almost negligible as seen in Figure 36.

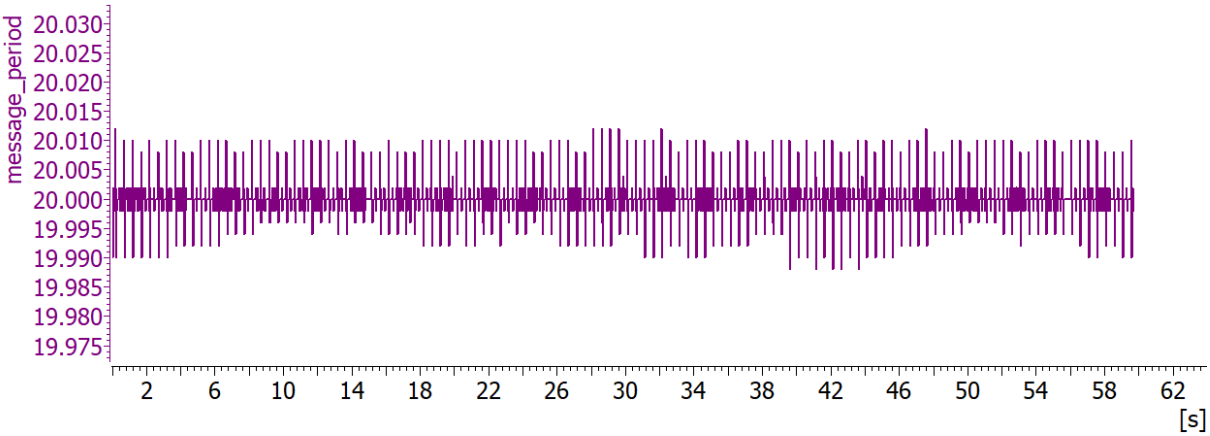

Figure 36 Periodicity measurement under no load in CAPL-on-board mode cycle time 20ms.

Table 16 lists some measurement values obtained from CAPL-on-board mode tests. Those message periods are 1ms, 10ms, 20ms, 50ms and 100ms. Independently of the chosen period, almost all statistical values are measured to be the same.

Table 16 CAPL-on-Board Periodicity vs Time measurements under no load.

Name	Period	Min	Max	Average	Standard Deviation
Periodicity	1ms	0.9840	1.01400	1.0000	0.0010
Periodicity	10ms	9.9840	10.0120	10.0000	0.0020
Periodicity	20ms	19.9840	20.0120	20.0000	0.0028
Periodicity	50ms	49.9860	50.0120	50.0000	0.0041
Periodicity	100ms	99.0880	100.0120	100.0000	0.0045

HIL with RTA experiments have also been done previously in literature. Figure 37 depicts a cyclic message of 20ms generated by a real-time adapter. CAPL-on-board mode of our test suite is comparable to the RTA or HIL extension of the literature. Although given names are different, the concept is the same. When statistical results are compared, our equipment again generated better average, min, max and standard deviation values.

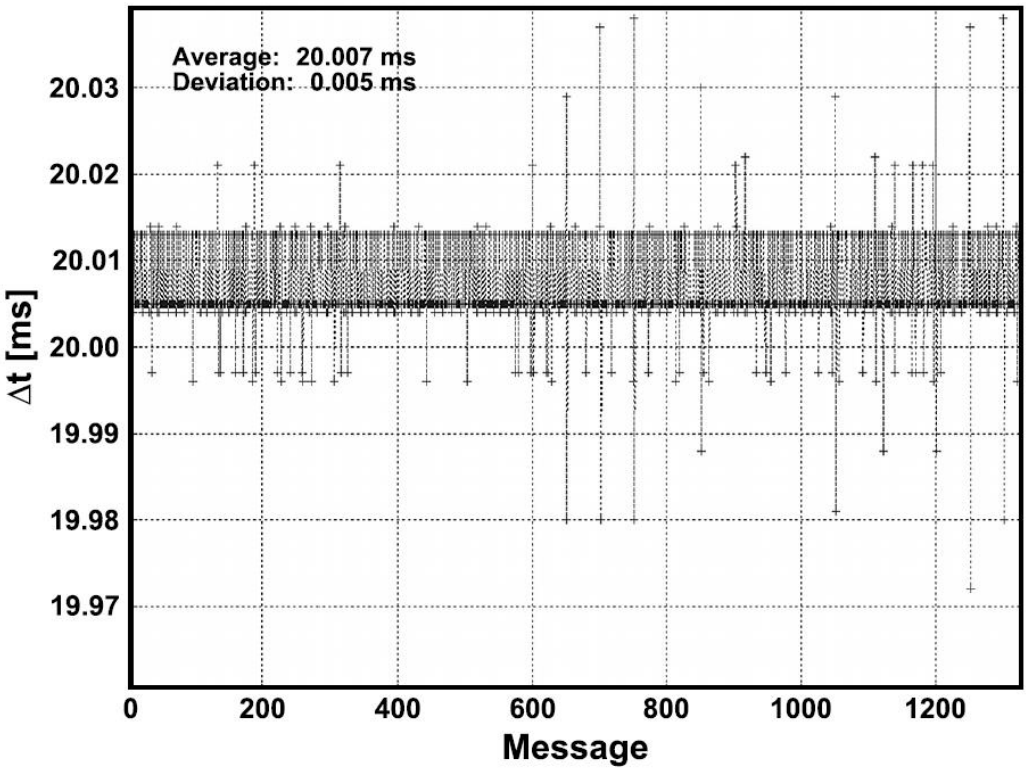

Figure 37 RTA (Real-time adapter) HIL for 20ms cycle time messages [38].

4.3 Rest-bus Simulation

Doing latency and periodicity measurement increased our confidence in the proposed solution. It is experimentally proven that it can handle periods as low as 5ms. Now it is time to simulate the rest of the bus. Since its inception in the 1950s in the aerospace

industry, HIL tests have been used in many industries including cars. Before the invention of HIL, manufacturers had to do their ECU test either on a real car or in an integration lab where individual ECUs are separated and connected to others. The downside of doing on-vehicle tests is that they can be impossible, dangerous, or damaging. The other disadvantages are that they are costly, time-consuming, and difficult to do. When a failure is detected at the on-vehicle test it needs to be handled. It is very hard to find the source of failure on-vehicle tests. Because all ECUs are interconnected and to find the failed one, one must do a lot of trial-and-error. If failure is confined to a single ECU, then it is relatively easy to fix it. However, if failure is related to an interface, then all ECUs using that interface need to be removed and fixed. These late fixes result in huge time losses. The failed ECUs need to be removed, their source code needs to be altered, unit tests need to be repeated and finally those ECUs need to be installed again. One simple failure causes a lot of backward progress. The way to solve this problem is front loading the tests. Engineers need to find a way to fix faults before they appear in on-vehicle tests. Thanks to the advanced software tools, we now have a way to test individual ECUs separately before they end up on a vehicle. It is called rest-bus simulation. Rest-bus simulation has been given many names in literature. Some of those names are residual bus, rest-bus, and remaining bus simulation. In this thesis we are going to stick to the rest-bus naming convention. HIL platforms became the de-facto testing environment for testing real ECUs. HILs allow testing real ECUs individually even though those ECUs need others to function. In today's vehicles, there are more than 70 ECUs interconnected. Software running on them usually depends on data originating from other ECUs. Those data are shared by the CAN interface. Sensors on the other hand are usually connected to IO pins and don't actively participate in the CAN network. Instead, they send their measurement values through their host ECU to the remaining bus. There are sensor variants that participate in the CAN. Rest-bus simulation only simulates CAN messages. Sensors that are connected to the IO interface of ECUs can't be simulated with rest-bus tools. They need additional hardware and software tools for simulation. One such hardware-software toolset is a data acquisition (DAQ) and LabVIEW toolset. DAQ devices are outside of the coverage of this thesis. In fuel cell systems there is only one or two ECU that manages the fuel cell stack. They have been given a special name FCU which stands for Fuel Cell ECU. In rest-bus application, our real ECU will be an FCU, and all the others will be simulated by software. FCU has 4 channels that allow it to participate in four different CAN buses. A CAN interface box used with our test laptop also has 4 channels which in turn will be used to connect corresponding channels of the FCU as seen in Figure 38. On the laptop side, behind each channel there exists virtual simulated nodes. Since those are simulated, it is possible to add as many nodes as

possible. Though there aren't any theoretical upper limits on the number of virtual nodes, a restriction emerges from memory and CPU capacity of the hosting computer. For our purposes a laptop with decent hardware is enough to do rest-bus simulation.

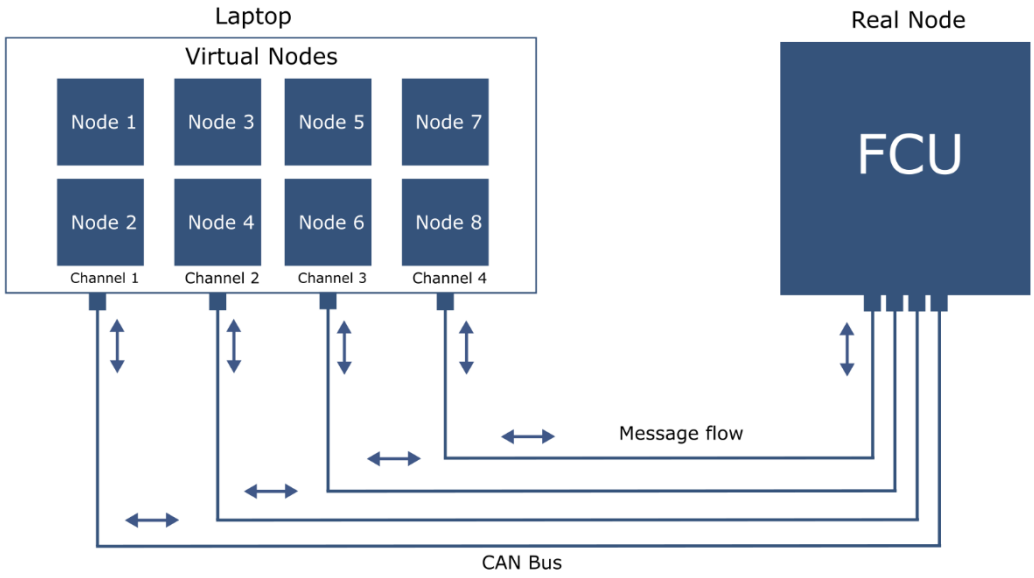

Figure 38 Layout of Rest-bus simulation.

With network interface hardware VN1630, 4 CAN channels of the box can be connected to FCU. The corresponding 4 channels of the FCU are named H2_OEM, FCS, HES and ENG respectively. Each CAN channel of the FCU is expected to be connected to relevant ECUs in a car environment. Our simulation setup above replaces a real car environment with a simulated one. The messages directed to the FCU from the virtual nodes must be simulated. The messages exchanged among virtual nodes don't need to be simulated. Once all nodes and their corresponding messages are put on the bus, an environment for a functioning FCU is created. FCU tests are essentially black-box tests. Only the reaction of the FCU to the outside world is evaluated. There are three ways to observe FCU's internal reaction. The first is through diagnosis interface, second is through calibration interface and the third one is by CAN interface. The Diagnostic interface interacts with the basic software of the AUTOSAR stack. DCM responds to diagnostic messages encoded in UDS protocol. UDS uses CAN as the underlying transport protocol. Faults are reported in DTCs. Through the diagnostic interface it is possible to read those generated DTCs. Calibration interface on the other hand allows reading/writing FCU's memory in runtime. It is used to watch and alter variable values in memory. Calibration interface uses the XCP protocol which again uses CAN as the underlying transport. Each CAN bus has a communication matrix created by the designers of the system. That database documents nodes, exchanged messages, periods, and signals.

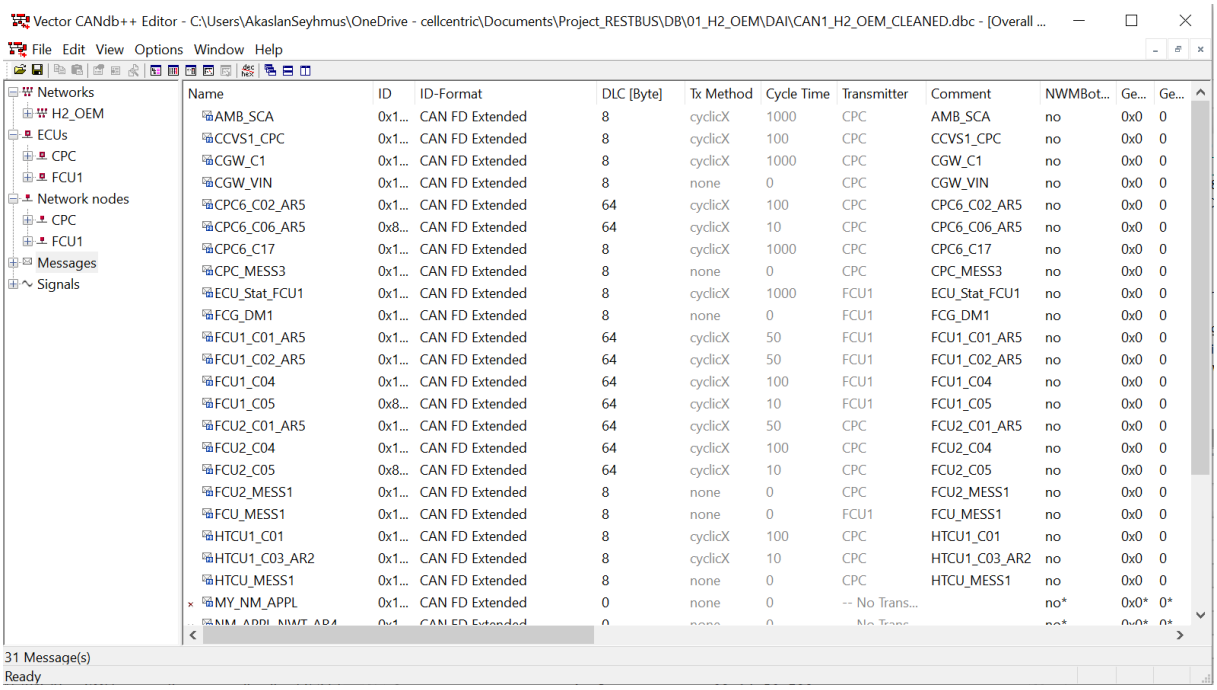

Figure 39 CAN1 Communication Matrix in DBC file format.

Once the communication matrix is ready it is possible to use it in CAPL code. According to the database in Figure 39 for CAN1 bus, only the CPC and Tester nodes need to be implemented. Figure 40 represents the layout of H2_OEM (CAN 1) and FCS (CAN-2). Simulated nodes are extracted from the database. TESTER nodes are there to provide network management related wakeup signals for the FCU. Actual functionality on the other hand is provided by remaining nodes.

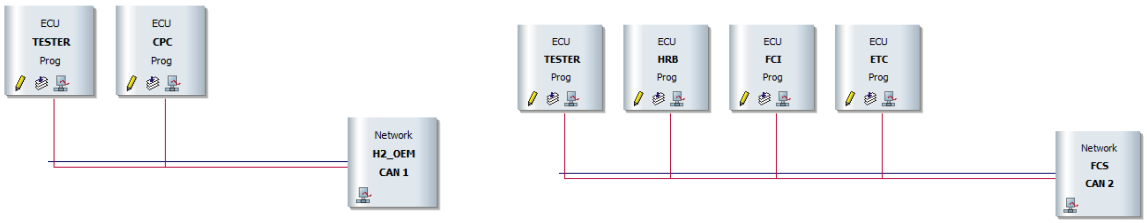

Figure 40 Simulation layout contains simulated nodes.

A part of CAPL implementation of the CPC node is given below in Table 17. Note that all other nodes have similar CAPL implementation.

Table 17 CAPL implementation of some timers and messages.

```
variables
{
  /* Timers */
  msTimer t_10ms;
  msTimer t_50ms;
  msTimer t_100ms;
  msTimer t_1000ms;
```

```
  msTimer t_noncyclic;

  /* Cyclic Messages */
  // 10ms
  message CPC6_C06_AR5 m_CPC6_C06_AR5;
  message FCU2_C05 m_FCU2_C05;
  message HTCU1_C03_AR2 m_HTCU1_C03_AR2;

  // 50ms
  message FCU2_C01_AR5 m_FCU2_C01_AR5;
  message SCA_C2 m_SCA_C2;

  // 100ms
  message CCVS1_CPC m_CCVS1_CPC;
  message CPC6_C02_AR5 m_CPC6_C02_AR5;
  message FCU2_C04 m_FCU2_C04;
  message HTCU1_C01 m_HTCU1_C01;
  message SRS_A2_AR2 m_SRS_A2_AR2;

  // 1000ms
  message AMB_SCA m_AMB_SCA;
  message CGW_C1 m_CGW_C1;
  message CPC6_C17 m_CPC6_C17;
  message TD_ICUC m_TD_ICUC;

  /* Non-Cyclic Messages */
  message CGW_VIN m_CGW_VIN;
  message CPC_MESS3 m_CPC_MESS3;
  message FCU2_MESS1 m_FCU2_MESS1;
  message HTCU_MESS1 m_HTCU_MESS1;
}
```

The variables section contains timers and message definitions. Event handlers are blocks of code that react to system events. Those events can be message receive events, keyboard events, system variable events and timer events.

Table 18 CAPL implementation of some event handlers.

```
on sysvar H2OEM::CPC
{
  if (@H2OEM::CPC == @H2OEM::CPC::On)
  {
    setTimer(t_10ms, 10);
    setTimer(t_50ms, 50);
    setTimer(t_100ms, 100);
    setTimer(t_1000ms, 1000);
    setTimer(t_noncyclic, 1000);
  }
  else
  {
    cancelTimer(t_10ms);
    cancelTimer(t_50ms);
    cancelTimer(t_100ms);
    cancelTimer(t_1000ms);
    cancelTimer(t_noncyclic);
  }
}
```

```
on timer t_10ms
{
  if (@H2OEM::CPC == @H2OEM::CPC::Off)
  {
    return;
  }

  setTimer(t_10ms, 10);

  fillMessageCPC6_C06_AR5();
  fillMessageHTCU1_C03_AR2();

  output(m_CPC6_C06_AR5);
  output(m_FCU2_C05);
  output(m_HTCU1_C03_AR2);
}
```

In addition to simulating nodes and generating messages, the CANoe tool also allows measuring signal values in the time domain and draws them on graphs. For demonstration purposes some of the signals in CAN1 messages are captured, and statistical information is extracted based on their periods. Figure 41 depicts some messages coming from the FCU as well as from the simulated CPC node.

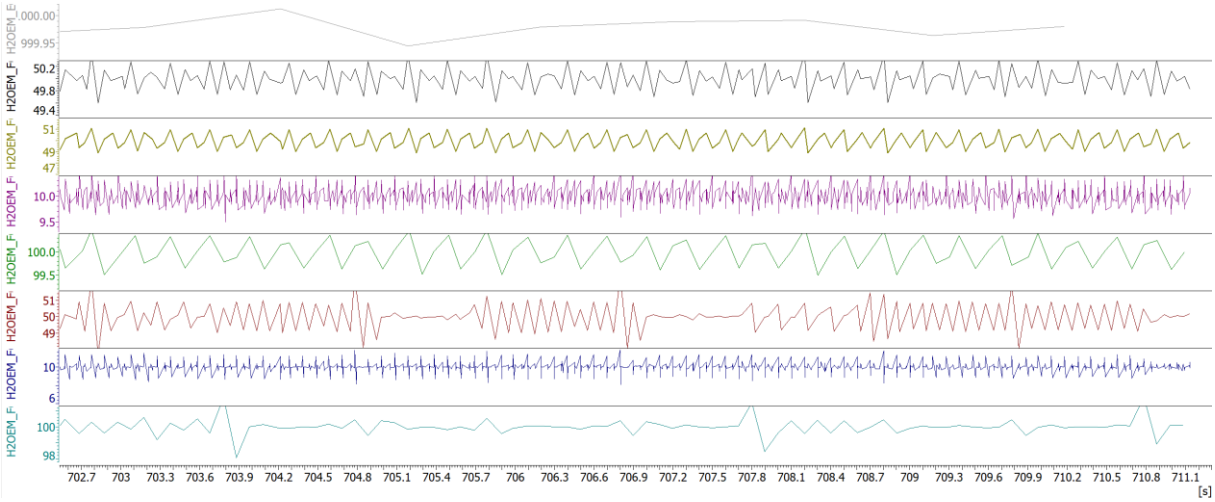

Figure 41 Some captured signals in rest-bus simulation.

Statistics obtained from the above graph is given in Table 19. Ones, whose TX columns are FCU, are generated by FCU, whereas other ones are generated by simulated nodes. What makes this table different from the results obtained in previous periodicity experiments is the number of messages present on the bus. Rest-bus simulation contains all the messages supposed to be available on the real vehicle bus. The total load of the CAN bus is bigger, approximately 40% compared to the periodicity experiments. For that reason, it is expected to observe deviations of signals bigger than previously observed ones.

Table 19 Statistical values of some captured signals in rest-bus simulation.

Name	TX	Period	Min	Max	Average	Standard Deviation
ECU_Stat_FCU1	FCU	1000ms	999.9445	1000.0125	9999.9767	0.0175
FCU1_CO2_AR5	FCU	50ms	49.5964	50.4253	49.9988	0.2015
FCU1_CO1_AR5	FCU	50ms	48.8998	51.1218	49.9988	0.6930
FCU1_CO5	FCU	10ms	9.5047	10.5902	9.9999	0.1890
FCU1_CO4	FCU	100ms	99.4995	100.4951	99.9937	0.2956
FCU2_CO1_AR5	CPC	50ms	47.5668	52.4503	50.0003	0.8603
FCU2_CO5	CPC	10ms	8.1075	11.8382	10.0004	0.5757
FCU_CO4	CPC	100ms	98.0652	101.8655	100.0005	0.4946

One obvious result that can be deduced from Table 19 is that simulated nodes have a little worse outliers compared to the real ECUs. Average values, however, are remarkably close to the planned cycle times. Simulated nodes have this behavior because of the increased busload shown in Figure 42. The more load the worse it gets. FCU participates in CAN3 only in listening mode. This is the reason behind the low bus load on CAN3. FCU receives the hydrogen sensor message HYS_01 from CAN3. In the gateway application next chapter this message will play an important role.

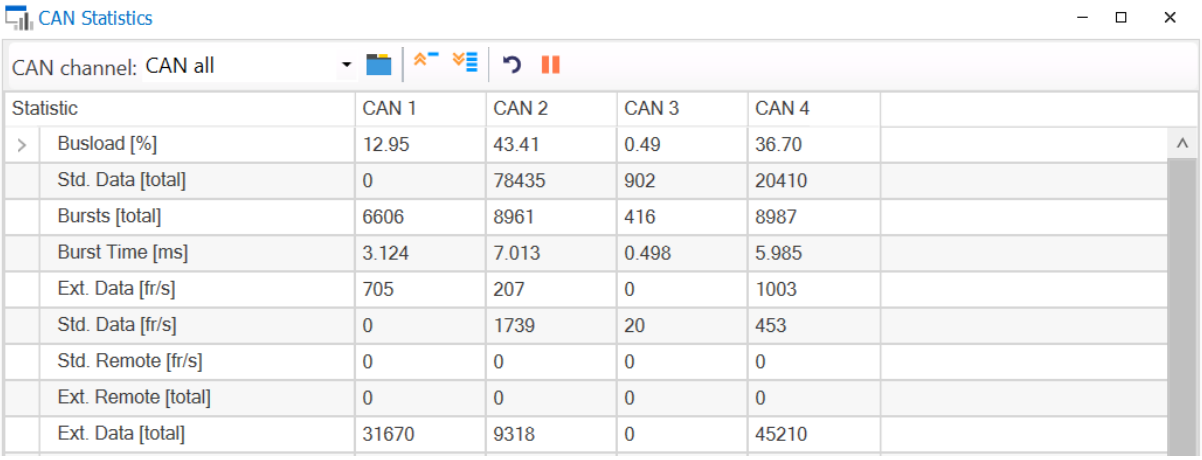

Figure 42 Statistics window of CAN interfaces.

Despite some extreme outliers, analysis of periods shows us average values of simulated signals remain very close to the intended values. Once simulation is started, messages start flowing in both directions. Figure 43 shows message flow between the simulation side and real FCU side. The figure is put for demonstration purposes and contains only CAN1 bus.

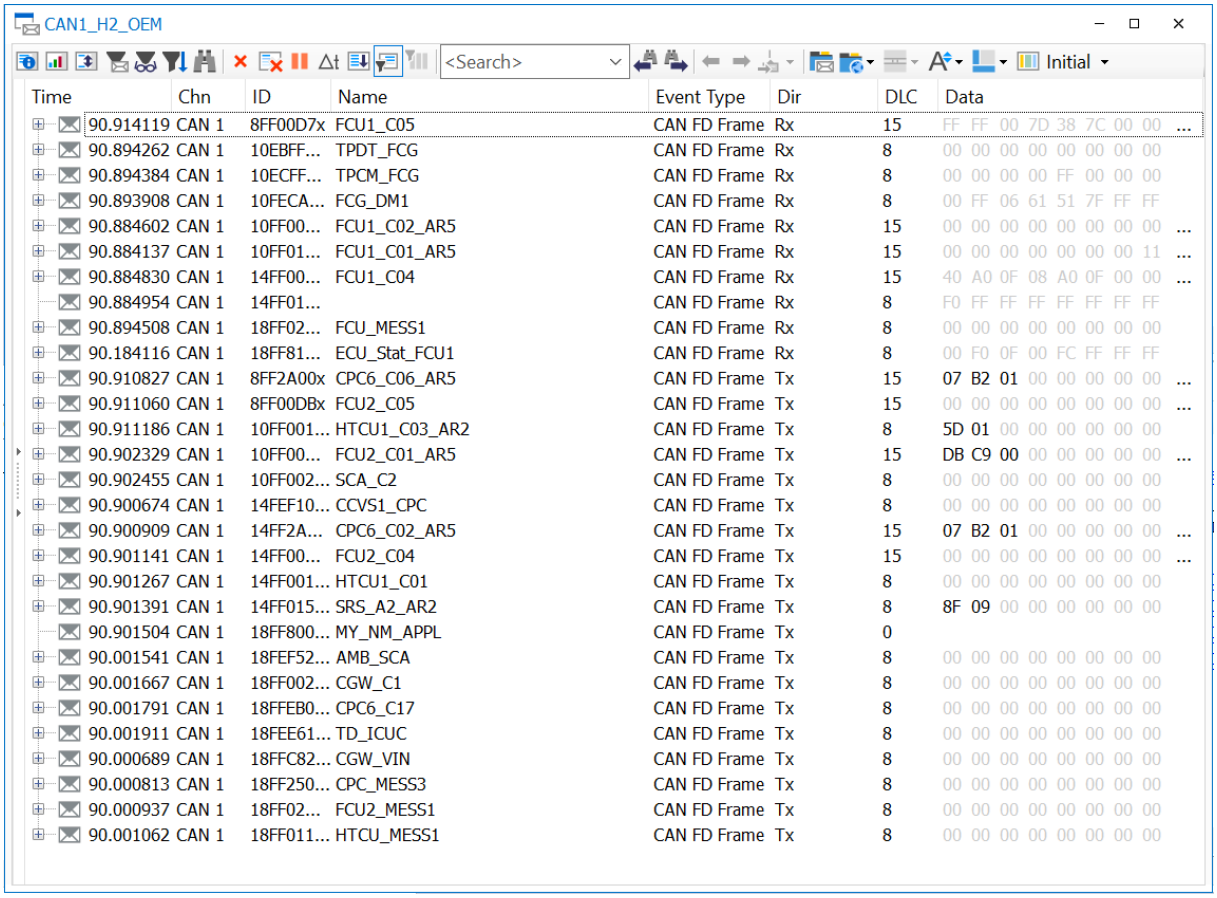

Figure 43 Rest-bus simulation CAN1 Data window.

CANoe compares and matches ID numbers of messages in the data window to the ID numbers of messages defined in the database. If there is a match, then that message's name is extracted and shown in the data window under the name column. This procedure is the same for all messages. The integration of the database reflects itself in other parts of the software as well. Once a message is identified its corresponding signal names are also extracted from the database and shown in the data window. In case there isn't any database connected to the project or the message definition missing in the database then all will be shown in the data window will be raw hex numbers. Seeing flows of TX (transmit) and RX (receive) messages are proof of a working simulation. TXs are the messages coming from simulated nodes, whereas RXs are messages originating from FCU. Seeing them both means simulated nodes and FCU is talking to each other.

Signals can be manipulated inside CAPL code or through panels. Code approach is favored when those manipulations are permanent or require less interaction to the user. Because of the safety requirements some messages are protected by extra fields. CRC and counter fields are examples of such fields. These extra fields need to be included into messages by simulated nodes. The best place to put that kind of

modification is the CAPL code itself. The alternative to the CAPL code is to design a panel and allow the user to enter extra fields manually. These panels ease interaction of users with the system. With the help of the panels signal values can be monitored, altered, or even disabled easily. Figure 44 shows some panels designed for signal manipulation. CPC6_C02_AR5 message has an ID of 0x14FF2A00 and generated by the CPC virtual node residing in CAN1. HYS_01 message on the other hand has an ID of 0x638 and is generated by HYS virtual node in CAN3. Both messages carry extra CRC and counter fields embedded in them.

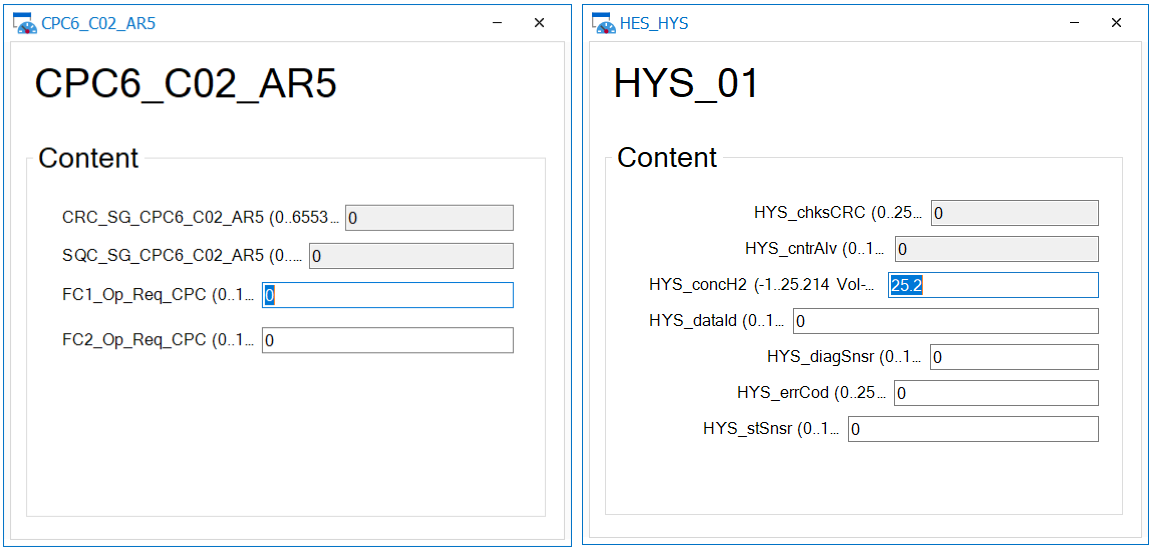

Figure 44 Panels created to ease message manipulation by hand.

Signal management is not the only use case for the panels. In addition to the previously mentioned use cases, panels can also be used for node management. Figure 45 shows a panel designed for node management. The panel has four groups corresponding to each CAN interface and node controls are laid out in each group.

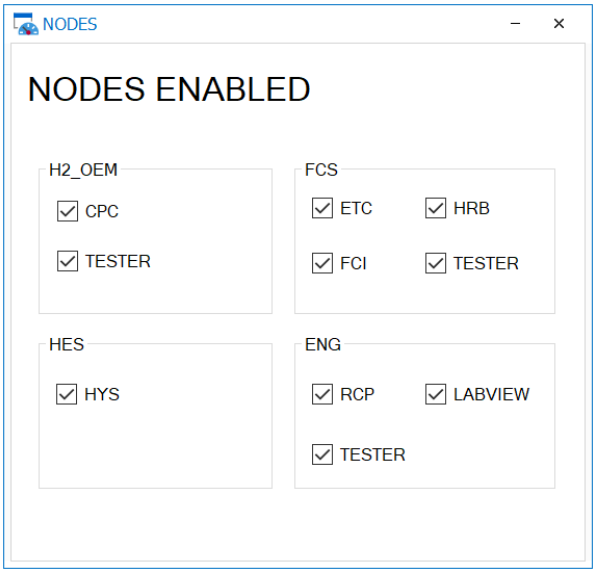

Figure 45 Panel allows easily enable and disable simulated nodes.

It is possible to do some tests on the messages while the simulation is running. When CPC, ETC and FCI nodes are disabled from the panel interface, corresponding DTCs (Diagnostic Trouble Code) are generated by the FCU. Those DTCs are listed in Table 20.

Table 20 Disabled nodes causes DTCs.

DTC	Description
EE0A80	CPC CAN Communication Timeout
1E0187	Electric Turbo Charger CAN Communication timeout
EE0787	FCI CAN Communication Timeout

The DTC column in Figure 46 contains numerical values corresponding to each DTC and the description column contains textual representation of the description. The status column on the other hand contains DTC status.

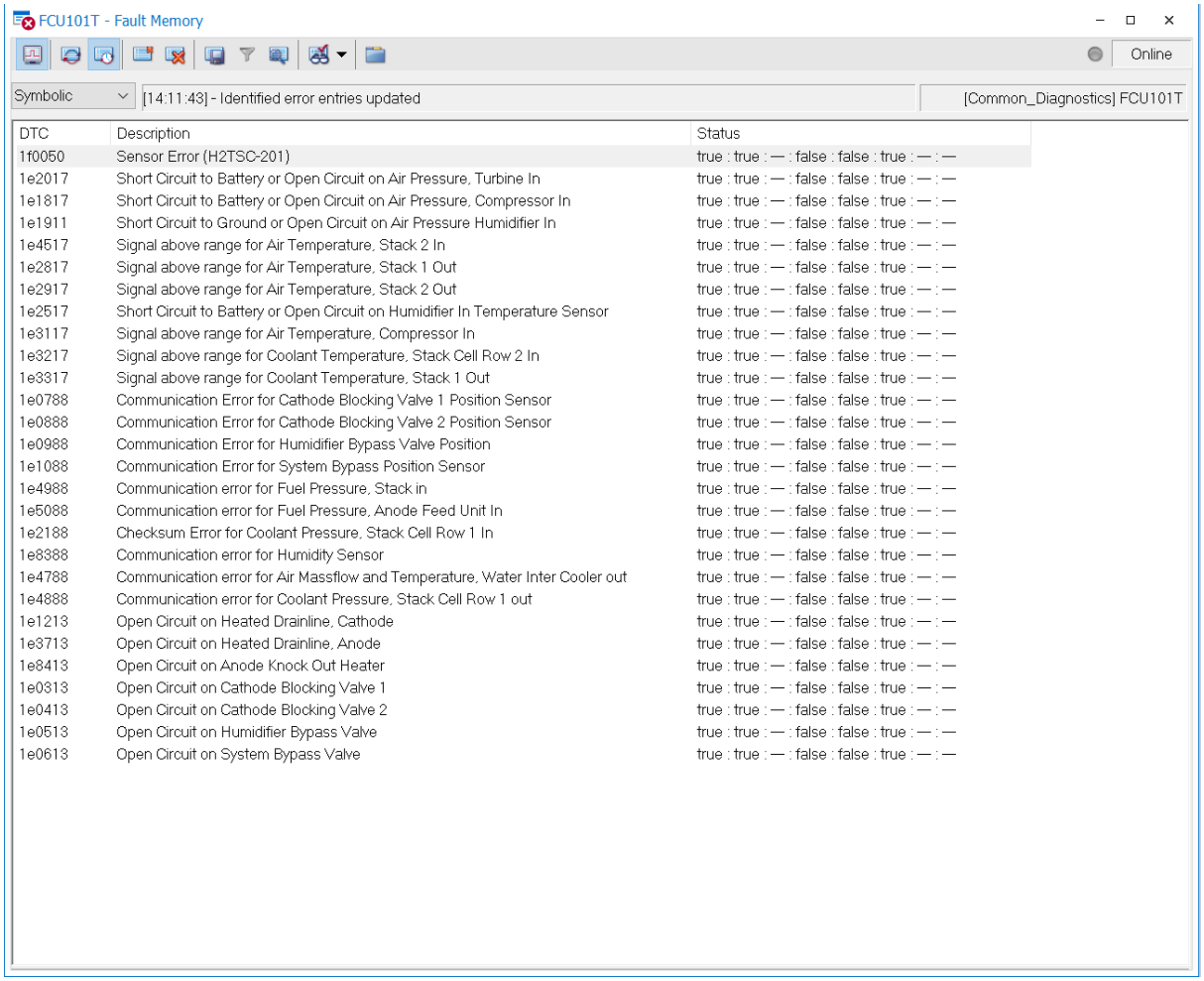

Figure 46 DTCs generated in the default state of the simulation.

Status is a bit-field with a size of 8 bits. Each bit corresponds to a different status. Bits can be true (1) or false (0) in each bit location. To get a list of DTC found in the following image a query with the $statusMask = 0x09$ needs to be constructed and fed into $ReadDTCInformation\ (0x19)$ service call. This query targets confirmed and failed DTCs. For details of bit statuses Table 21 can be referenced [36].

Table 21 DTC Status Byte [36]

statusOfDTC	Bit
testFailed	0x01
testFailedThisOperationCycle	0x02
pendingDTC	0x04
confirmedDTC	0x08
testNotCompletedSinceLastClear	0x10
testFailedSinceLastClear	0x20
testNotCompletedThisOperationCycle	0x40
warningIndicatorRequested	0x80

Figure 46 has listed some DTCs in working simulation because our experiment is not a perfect simulation. There are unplugged IO ports on the FCU side. This HIL setup can't simulate sensors connected to those unplugged IO ports. Rest-bus simulation only needs a CAN interface, and our simulation platform only connects to CAN pins of the FCU and can't simulate sensors on those missing pins. This is the main drawback of our proposed solution. Those sensors can only be simulated in real HIL benches. Established RTOS HIL simulators have additional hardware and software simulators. On those systems the DTC list in Figure 46 needs to be empty. For the sake of this experiment, we will ignore the DTCs generated by absent sensors supposed to be connected to IO pins. Apart from some DTCs, unplugged pins also create further problems. Some error check routines require those sensors present in the relevant pins. Since those conditions are never met, the DTCs related to other errors we hope to see on the fault memory will never show up. For example, HES sensor DTC is missing in the list. When we manually disabled HES node on panel, we hoped to see HES sensor timeout DTC. Since some prerequisite conditions aren't met due to the missing pins, HES sensor timeout DTC isn't generated at all.

Tests can be run manually or automatically. CANoe has testing tools which facilitate executing tests in an automated way. In this thesis both approaches are practiced. Having said that gateway tests are only done manually. Test scripts can be written in CAPL language. CANoe testing framework also supports C# programming language. Test scripts written in CAPL are fundamentally different from node simulation scripts. Unlike simulation scripts which run in an event-oriented fashion, test scripts run in a sequential fashion. Each block of the test case is atomic by design and can't be interrupted. Test scripts allow us to automate test execution. Automation brings speed, precision, reliability, and repeatability. Automation increases testing speed compared to manually executed tests. Testing time can be reduced to minutes compared to hours when done manually. Timing precision is also increased in automated tests. In scripting, durations in milliseconds can be controlled easily, whereas in manual tests only seconds can be controlled. Some conditions can only be satisfied in automated tests as such where multiple variables or conditions must be satisfied at the same time. In some situations, manual tests might be dangerous or even impossible. Human factors are always the source of errors. Automation removes human factors and increases reliability of the system. Once an automated test finds an error, it is very easy to repeat the same test and find the same error again. This makes it easier to find those errors that are hard to recreate conditions. Repeatability increases coverage and hence quality.

Figure 47 shows the test setup of the project. Test modules are grouped according to their functional, non-functional nature and CAN interfaces. All functional FCS tests, for example, are put under FCS root element. Each group contains functional test cases targeted for them. Non-functional tests such as load, stress, electrical tests are not part of this thesis. Functional tests include existence checks, period checks, DLC checks and DTC checks. This thesis will only cover functional tests. CANoe eases creation of functional tests by providing the test service library. Test service library is part of the Test Feature Set integrated into CANoe. Test service library has many predefined ready-to-use check, query, and statistical methods [27].

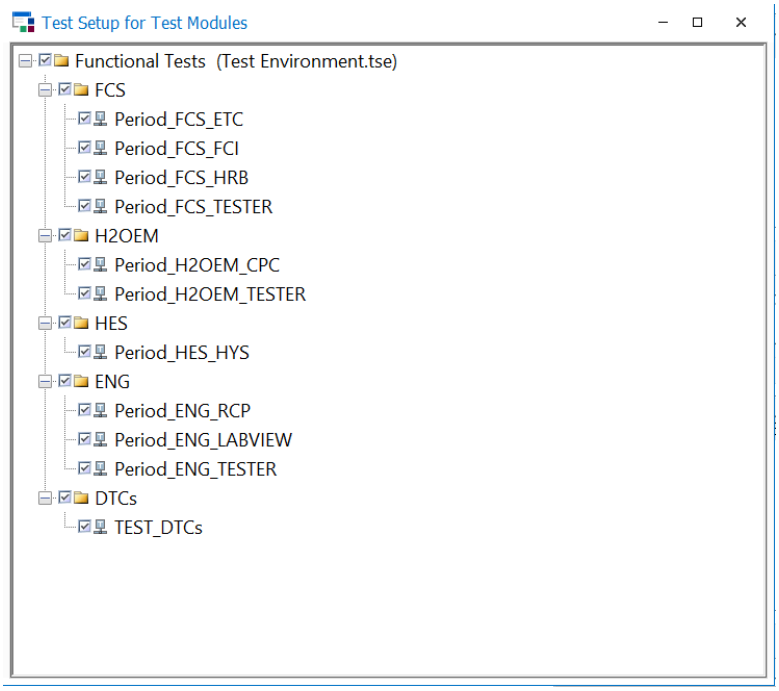

Figure 47 Test case grouping and setup window.

Simulated rest-bus nodes and FCU put a lot of messages into corresponding CAN channels. It is possible to analyze the system by just watching the data window and graphs tools of CANoe. This approach is what is called a manual test. We want to improve this by introducing test scripts and automation. For this end, a test case is created for each message. Test case contains checks for message existence and period measurements. Message existence checks, looks for a message in the bus for a given duration of time. If it is found then the test step will pass, if not the test step will fail. Period measurement tests listen to the bus for a given duration and measure periods between successive messages. If the observed period is within given limits, then the test step will pass, or else the test step will fail. Since the number of messages is too high on each bus, we won't be able to list them all here. However, for the demonstration only, one test case checking period and one test case checking DTCs

will be shown. Table 22 shows a test case checking the existence and the period of the ETC_Data message.

Table 22 A sample test case showcasing existence and period check.

```
testcase TC_ETC_Data()
{
  char buffer[100];
  const long cycle = 10;
  const double tolerate = 40;
  const long cycleMin = cycle * (1 - tolerate / 100);
  const long cycleMax = cycle * (1 + tolerate / 100);
  const long wait = 2000;
  dword id;
  double avg;
  double min;
  double max;


  id = ChkStart_MsgAbsCycleTimeViolation(ETC_Data, cycleMin, cycleMax);
  testAddConstraint(id);
  testWaitForTimeout(wait);

  avg = ChkQuery_StatProbeIntervalAvg(id);
  min = ChkQuery_StatProbeIntervalMin(id);
  max = ChkQuery_StatProbeIntervalMax(id);

  if(ChkQuery_NumEvents(id) > 0)
  {
    snprintf(buffer, elCount(buffer), "Valid values %dms - %dms", cycleMin,
cycleMax);
    testStepFail("", buffer);

    snprintf(buffer, elCount(buffer), "Avg cycle time: %.2fdms", avg);
    testStepFail("", buffer);

    snprintf(buffer, elCount(buffer),"Min cycle time: %.2fms", min);
    testStepFail("", buffer);

    snprintf(buffer, elCount(buffer),"Max cycle time: %.2fms", max);
    testStepFail("", buffer);
  }
  else
  {
    snprintf(buffer, elCount(buffer), "Valid values %dms - %dms", cycleMin,
cycleMax);
    testStepPass("", buffer);

    snprintf(buffer, elCount(buffer), "Avg cycle time: %.2fdms", avg);
    testStepPass("", buffer);

    snprintf(buffer, elCount(buffer),"Min cycle time: %.2fms", min);
    testStepPass("", buffer);

    snprintf(buffer, elCount(buffer),"Max cycle time: %.2fms", max);
    testStepPass("", buffer);
  }
```

```
  testRemoveConstraint(id);
  ChkControl_Destroy(id);
}
```

Test scripts start executing after a click of a button. Each test module listed in Figure 47 contains a lot of test cases. When a test module is executed all its test cases defined inside run one after another. Depending on the result, a test case ends up either a pass or a failure. Failed as well as passed test cases in each module are reported after test execution. Figure 48 shows the execution result of Period_H2OEM_CPC test module and its test cases. Test cases are executed one after another in sequential order. The verdict column shows that all test cases are passed, and the runtime column shows time taken for each one of those cases. Total time spent is shown beneath the panel which is 41 seconds. Test modules are nothing but regular CAPL scripts. The special thing about test scripts when compared to regular scripts is that they run in sequential manner without interruptions. Every statement in a test script will be executed one after another without interruption until the script calls a wait statement. Wait statements are the only place script transfers execution to other threads. On the opposite side, regular CAPL scripts run in event-driven fashion. Every event triggers an interrupt in the normal flow of execution [27].

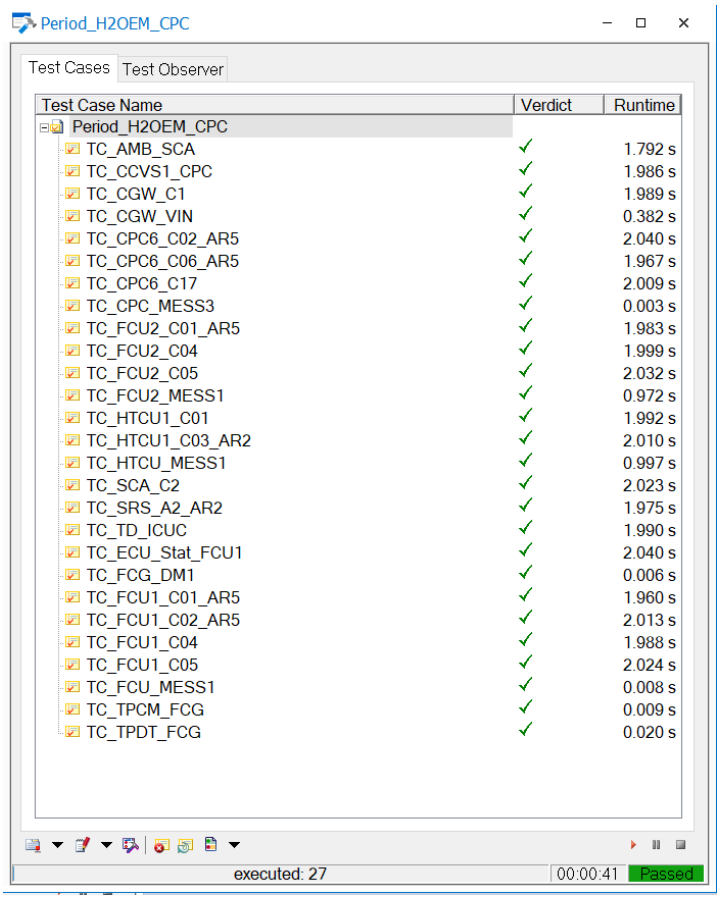

Figure 48 A test module and its test cases are executed.

Test modules themselves can be either executed one at a time or all at the same time. Total execution time for all modules is measured to be around 3 minutes. This value is exceptionally good when compared to manual tests which take at least a week to finish. Automatic test report generation is another feature provided by CANoe. The report contains test results of each test case as well as extra information outputted by test scripts.

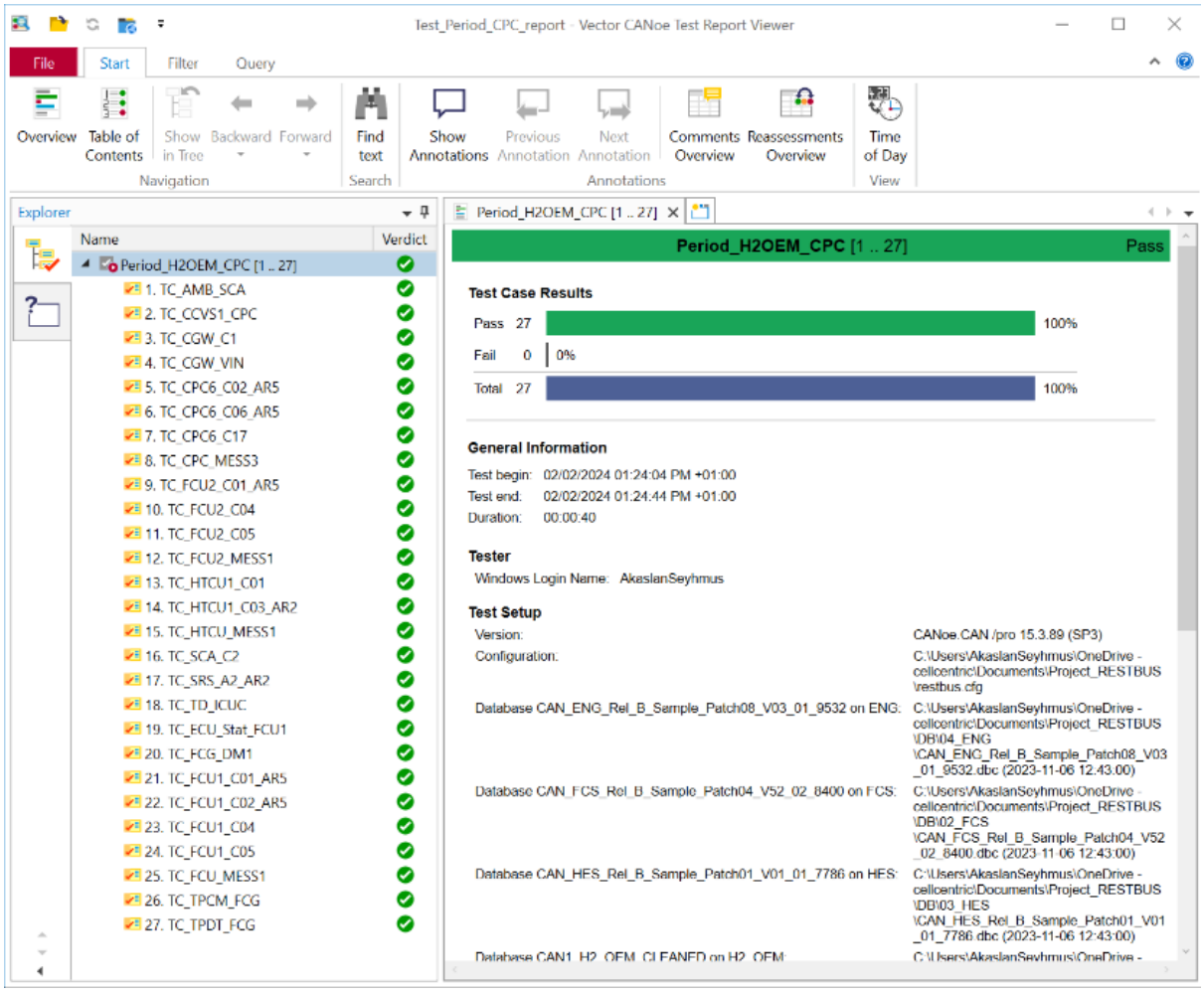

Figure 49 The test report is optionally generated after test execution finalize.

4.4 Gateway Application

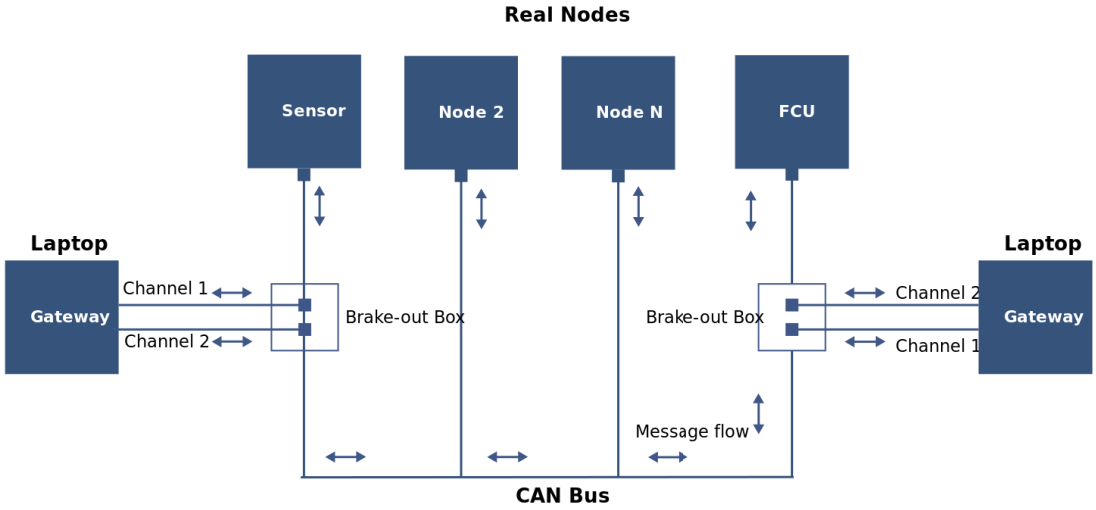

Figure 50 Overall layout of Gateway simulation.

The idea behind the gateway application is to cut the bus line into two, plugin the gateway in between and intercept the messages flowing in both directions. This intrusion enforces CAN traffic to pass through gateway. Gateway laptop now has full control over all messages passing over it in both directions. Enforcing message traffic to pass through the gateway opens many possibilities. Message relaying, message filtration, message creation, signal modification are among those possibilities [40]. Apart from mentioned ones, reverse engineering CAN messages and trying to remanufacture existing ECUs are also among other use cases [41]. Gateway laptop can be installed in two possible locations as seen in Figure 50. Our purpose in this experiment is to modify hydrogen sensor signal values before they reach the FCU. Gateway can be positioned either just after the sensor or just before the FCU. Placement of gateway in both locations has its advantages and disadvantages. However, we found out that it was difficult to place the gateway just before the FCU in the system test bench because of accessibility reasons. The hydrogen sensor on the other hand was outside of the fuel cell stack in an easily reachable position. Depending on the placement of the gateway, information we are going to observe on each channel will be different. If gateway is placed just after sensor, channel1 will show only sensor messages coming from sensor. Gateway will relay that message to the remaining network by putting it on channel2. Apart from relaying, the gateway has freedom to block, modify or even create its own sensor messages. Whereas in the second case, gateway is put just before the FCU. In this placement channel1 contains sensor messages in addition to messages coming from the rest of the network. Channel2, however, contains only messages coming from FCU. This placement has an

advantage when it is desired to change FCU signals before they reach the rest of the network or even try to determine which messages belong to the FCU if it was unknown beforehand. Gateway may do many operations on messages before relaying them over. Figure 51 illustrates a panel designed to manage gateway operations. The top part of the panel contains two modes. Either relay mode or simulate mode can be selected at one time. The bottom part of the panel contains fault injection methods. In the center of the panel signal override functionalities are placed.

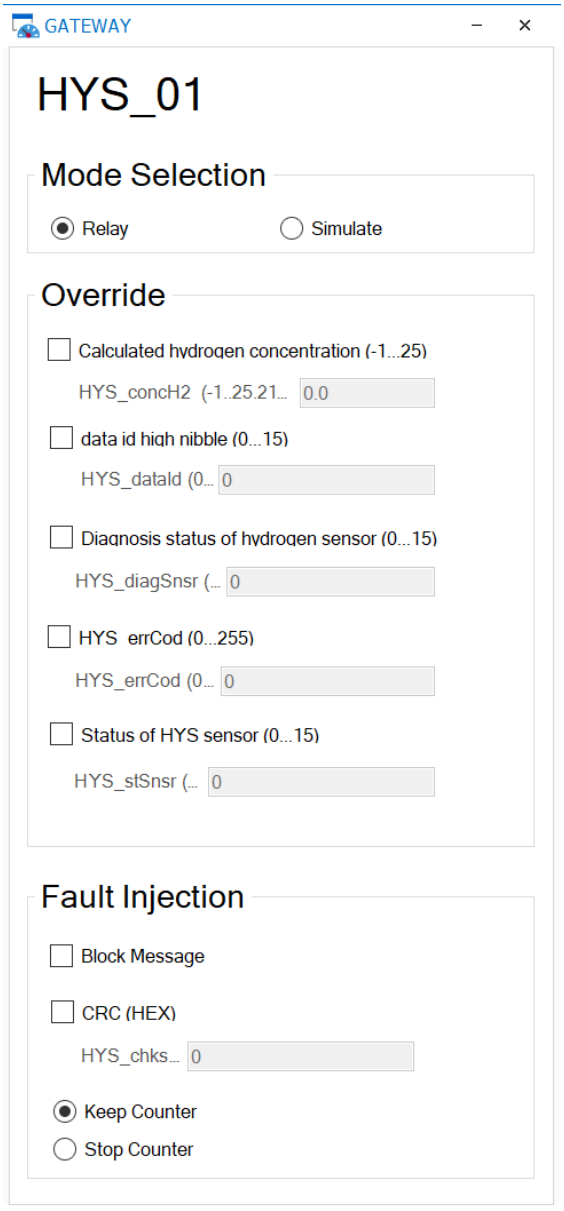

Figure 51 Panel allows easy management of sensor signals.

Relaying operates in both ways. In relay mode, gateway relays all messages arriving in its first channel to its second channel. By design messages arriving in channel2 are untouched and relayed to channel1 behind the scenes programmatically. Intention here is to capture sensor message and depending on use case alter it or relay it as it

is. If we wanted it to pass the sensor message untouched then the tick boxes on the Override section of the panel won't be selected for any field. If the intention is to modify some fields, then corresponding fields can be selected on the panel and the user can decide what value to put into those fields. Those fields on the override section of the panel correspond to one signal. Override and fault injection sections of the panels are shared between relay and simulation mode of the gateway. The fault injection section of the panel contains functionality to block the message or change its CRC and counter fields.

In simulation mode, gateway filters sensor messages arriving at its first channel and blocks them. Gateway creates its own sensor signals programmatically and forwards them to its second channel. The intention here is to replace original sensor messages with simulated ones. Messages put on its second channel will make their way to the FCU. This capability to simulate sensor signals might come in handy when a real sensor is not available yet.

Override section is shared by simulation and relay modes. Signals values can be modified before relayed to the second channel. Once the desired message arrives on the first channel, its signal values can be modified according to the user's wishes and are forwarded to the second channel. All other messages will be untouched. Modification is done in one direction only. Messages arriving on its second channel will be forwarded to its first channel untouched.

The bottom part of the panel contains fault injection methods. The first control is for message blocking, the second one is for CRC alteration, and the last one is for stopping the counter.

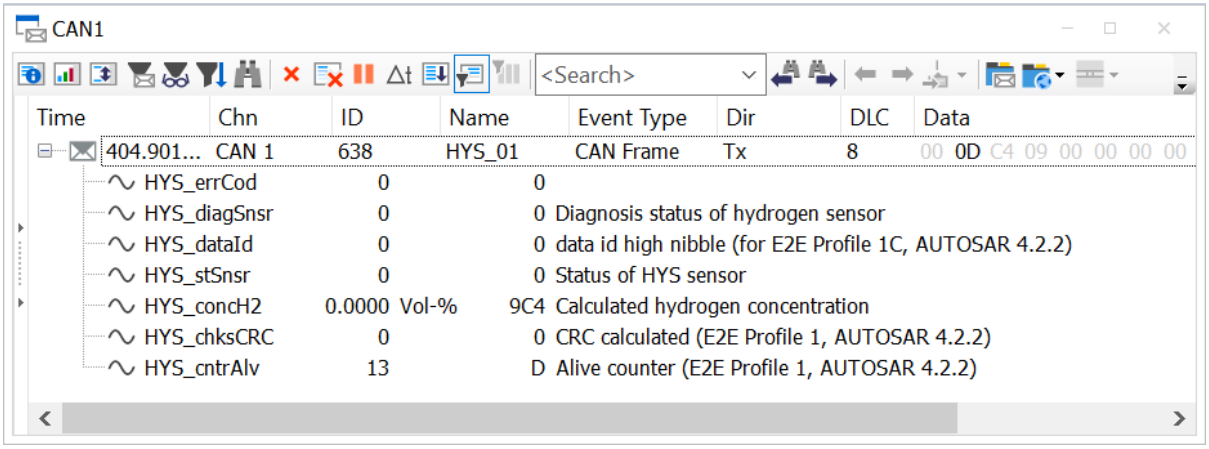

Figure 52 Data view of incoming sensory messages.

Target message HYS_01 is visible on the data window of channel1 as shown Figure 52. This message is an original message transmitted by the sensor. Our gateway interrupted transmission of this message before its arrival at the destination. Channel1 data window shows incoming messages and channel2 data window shows relayed messages. Data window belonging to channel2 is shown in Figure 53. A quick look at both images reveals differences between the same signals. For example, hydrogen concentration is modified to 2.0, sensor status is 1 and data ID field is now 15. In case fault injection methods are used those would be visible too. Stop counter and CRC alteration were among those methods that can be spotted at relayed messages easily.

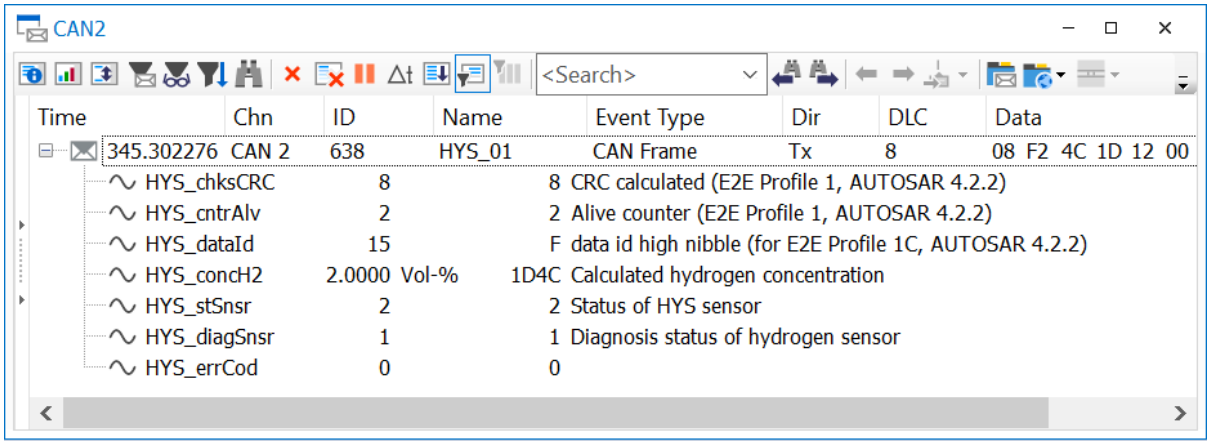

Figure 53 Data view of outgoing altered messages.

5 Results and Evaluation

Chapter 2.4 is dedicated to mathematical worst-case scenario calculations for CAN frames. The transmission speed of a classic base frame and classic extended base frame are calculated to be 0.2636ms and 0.3125ms respectively. For the calculations bus bitrate speed is assumed to be 512Kbps. FD frames introduce higher speeds and larger payloads. Similar transmission speed calculations are done for FD frames too. It is assumed that the bitrate switch (BRS) flag is set in FD frame, and the high bus speed is 5Mbps. Under these assumptions transmission speed of an FD extended frame is calculated to be 0.2435ms. One conclusion that can be drawn from this section is that transmission speed varies in range $[0.2435ms, 0.3125ms]$ for all kinds of frames. Since this range is very narrow, for practical purposes transmission speed of all kinds of frames can be rounded 0.3ms.

Chapter 4.1 is dedicated to investigating latency of messages in the simulation platform. The objective of this section was to determine the limitations of the proposed platform by examining inherent latency. The latency experiment is done to measure the total transmission time of a classic base frame. Unlike theoretical calculations done at Section 2.4, measurements in this section included processing time spent on nodes at both ends. Processing time of nodes can be influenced by many factors such as the non-real-time nature of the OS, background processes, antivirus software, keyboard events, paint events, mouse events, just to name a few. Measured latency mean values observed to slide between $[0.6199ms, 1.5123ms]$ range and observed max outliers was up to 3.43ms under a lightweight bus load. The sliding nature of average value was a very surprising finding, and it is attributed to the non-real-time nature of the OS. If we assume that an average transmission time from node to node is 1.4847ms in an empty bus and subtract theoretical transmission time spent on the bus (0.3ms) from this value, we will get processing time of a node $node\_time = (1.5123ms - 0.3ms)/2 \cong 1.2ms\ /2 = 0.61ms$. The amount of 0.61ms is the best average time spent on a virtual node to process a message in an empty bus. It is hard to predict the performance of the solution on a very busy bus load. In an empty bus 3.43ms max outlier is measured according to Table 8. But this value is expected to increase depending on the bus load. In the max outlier case node processing time is $node\_time = (3.43\text{ms} - 0.3)/2 \cong 1.57\text{ms}$ and this puts a worst-case processing time on the simulation platform. Although CANoe can generate 1ms timers, according to tests done in this section processing time of 1.57ms is too big for a 1ms message. Percentage wise speaking outliers are too big compared to 1ms period and this kind of deviation might not be tolerable for some systems. For safety-critical applications

which require latencies smaller than 5ms, this solution platform shouldn't be used. This conclusion is reached by examining max outliers found in Table 8 and Table 11 in latency experiments.

Likewise, Chapter 4.2 is dedicated to investigating periodicity of messages in simulation platforms. The objective of this section was to determine the limitations of the proposed platform by examining period violations. While in periods bigger than 1ms, outliers' deviation compared to the average period was negligible, it was strikingly big for 1ms messages. Simulation platform barely copes with messages of 1ms even in empty buses. Measured max outlier according to Table 14 is 1.7460ms. Underlying reasons caused latency deviations are at play even for period fluctuations. For a loaded bus, the situation gets even worse. According to Table 15 max outlier for 1ms periodic message is 2.6780ms. This is two and half times more than the period itself. Fortunately, there weren't any periodic messages less than 10ms in the communication matrix of the FCU under test. Also, outliers didn't cause any diagnostic trouble codes either. The other important observation of this experiment was that no matter which period is used, the testing solution generated impeccable average values regardless of bus load. When the experiment was repeated with CAPL-on-board mode, the max outlier observed was around 0.1ms. The resulting deviation was so small because of the real-time nature of the hardware platform. Overall, obtained results are aligned to periodicity experiments done earlier in literature.

Chapter 4.3 is dedicated to rest-bus simulation of the entire network. Apart from the FCU, all other nodes participating in the CAN bus are simulated. According to Table 19, comparatively speaking min, max deviations observed in virtual node messages are bigger than the FCU messages. This result was expected because FCU is a hard real-time capable device, while simulation platform is not. Average measured periods on the other hand are impeccable on both sides. Automated tests are used to measure signal periods as well as stimulate the system. FCU's reaction is observed in the fault memory list for some external stimuli exerted by modifying signals on the CAN bus. Deviations more than %40 percent from the mean value are observed in rest-bus simulation while doing automated tests. This value is more than what is measured in Table 19. One conclusion that can be drawn from this finding is that the automation process itself generates additional load on the system and this further increases deviations. Overall, the proof-of-concept implementation of a rest-bus simulation on low-cost hardware has been proved to be successful.

Chapter 4.4 is dedicated to the gateway application. Gateway applications are very helpful when message alteration tests need to be done on actual hardware either on a system bench or on a vehicle. For example, too much hydrogen concentration creates a potentially dangerous situation, and it needs to be tested. Using real hydrogen gas to increase concentration would be dangerous. Instead, a gateway application is used to increase hydrogen concentration level more than %2 percent by modifying sensor output and relaying it to the rest of the bus. If the system shuts down, it means FCU is working correctly. If not, then it means FCU isn't working according to the specifications. HYS_01 message in CAN3 was the target message. The solution was lightweight and easy to carry around due to its portability. System performance was also previously proved to be sufficient in simulation and periodicity tests. Overall, the proposed solution proved itself to be more than enough to satisfy requirements.

One of the original motivational points was to reduce queues accumulating in front of the HIL benches. Front loading is believed to be the solution to this problem and low-cost alternative HIL solutions which could be stationed on a desktop are investigated. Previous assessments proved that such a solution is feasible and in fact works satisfactorily. The other motivational point was to reduce testing time. Testing time is reduced partly due to front loading HIL tests onto the developer's desktop. Developers can fix faults before they make their way to the HIL bench. This effectively reduces the number of test iterations waiting to be done at the HIL bench. But how much time is saved by doing a test on a desktop HIL instead of a HIL bench? Time spent on doing one test on the HIL bench takes approximately 2.5 days. This amount includes preparations of the platform and executions of tests. On the other hand, time spent doing one test on a desktop HIL takes approximately half a day. Again, this time includes both preparations and executions. Times mentioned here are educated guesses based on previous experiences and regarded as reasonable. On a desktop HIL, it is shown that execution of all tests took approximately 3 min to complete in automated tests. Overall, it is safe to assume that time saved on testing by the proposed solution is around $2.5\ days\ -\ 0.5\ days = \ 2\ days$ for each test iteration.

And how much testing cost is eliminated? Reducing testing cost was also a motivational point. Total time spent on executing all tests on the proposed solution is assumed to be around half a day. This value includes preparation of the test platform, mounting cables of FCU to the laptop etc. Suppose the hourly rate of a developer is 120 Euro. On the contrary, the same tests are assumed to take 2.5 days in HIL bench. Total time spent on HIL for one test including preparations is around 2.5 days and

hourly rate for a test engineer is assumed to be 120 Euro. Suppose the same test is repeated (n) times for the duration of the project due to failure.

$$Total\ Cost\ of\ doing\ (n)\ tests\ on\ HIL = 2.5\ days * 8\ hours * 120\ Euro * n = 2400\ Euro * n \quad (8)$$

$$Total\ Cost\ of\ doing\ (n)\ tests\ on\ Desktop = 0.5\ days * 8\ hours * 120\ Euro * n = 480\ Euro * n \quad (9)$$

For one test the difference and hence total savings is $2400\ Euro - 480\ Euro = 1920\ Euro$. This value gets increased for every iteration. A particular benefit of doing HIL tests on desktop is the increased parallelism and testing frequency. Since desktop HIL is readily available to all developers whenever they need, tests can be repeated more than once. This results in the cost difference getting even bigger.

Front-loading tests before the HIL has additional benefits. Because we are eliminating faults earlier when they are cheaper to fix, the total cost of the project also decreases. However, total cost savings on the project level can't be measurable easily. The work done in this thesis can only measure testing costs.

Testing frequency is also increased because developers can repeat tests as much as they wish. Increase in testing frequency equates to less defect density. This in turn means better quality software.

Both manual and automated testing is practiced throughout the chapters. Automation brings a lot of benefits to the table compared to manual tests. Some of those benefits are increased testing speed, flexibility to run tests anytime, increased coverage and reduced human effort. Due to being flexible, scripts can be executed even off times. Likewise, automation removes human factors and can do near to impossible regression tests, stress tests, memory-leak tests, security tests. With the ability to recreate faulty conditions and increased testing coverage, undetected defects decrease even further, and hence quality level climbs higher.

6 Conclusion

Although HIL benches have many benefits there is still room for improvement. The desire to shorten the queues built up behind the HIL benches, reduce testing times and testing costs motivated us to search for alternative HIL platforms. It is believed that front-loading is the solution to this problem.

Proof-of-concept implementation of RBS and the gateway application were a success. One important statement must be made here. COTS based solutions can't replace entire established HIL platforms. Although COTS solutions have a lot to offer, they still don't have all the functionalities offered by the expensive systems. One such functionality is the non-functional tests such as electrical open circuit test, short to ground tests. The other limitation of the proposed was inability to simulate other peripherals due to some unplugged IO ports. Solution can only simulate CAN messages plugged into the CAN IO pins. Sensor and actuator signals plugged in to the remaining IO ports can't be simulated. The proposed solution is in a sense an incomplete HIL simulator but on the bright side this solution is best suited to gateway applications. A laptop alone is more favorable than heavy or hard to carry solutions.

Bringing a small-scale low-cost HIL solution to every developer's desk is expected to increase parallelism and hence reduce queues. Testing time has proved to be reduced from 2.5 days to 0.5 days, testing costs likewise proved to be reduced from 2400 Euros to 480 Euros for each iteration. Due to readily availability on each desk, testing frequency is increased. Increased testing frequency is also a requirement in agile methodologies. Testing time and undiscovered faults are inversely related. Increase in software testing frequency brings down the number of undiscovered faults and hence increases software quality. Fixing crucial faults while they are easier and cheaper to fix also improves the overall project cost.

It is also shown that the proposed solution ideally should be used in applications where signal periods are 5ms and above. This limitation originates from deviations on periods and inherent latency due to the non-real-time nature of the proposed solution. If the system under test can't tolerate up to 5ms latency or period deviations, then this solution isn't appropriate.

Periodicity experiments done earlier in the literature on COTS laptop and RTA platform are aligned with results obtained in this thesis. Overall, max deviations in periods

observed in our tests were less than the deviations reported in literature. This result was attributed to software and hardware improvements.

For future work, inclusion of the plant model directly into virtual nodes can be investigated. The ability to put plant models directly on HIL simulation is believed to bring many benefits. In model-based development (MBD) plant, controller and test models are developed according to customer requirements. Reusability of plant and test models at different testing levels increase consistency, expected to eliminate many script repetitions and mistakes along the way. Currently each testing level such as SIL and HIL must prepare their own test scripts. Automatic generation of test scripts instead of manual preparation is the idea behind MBT (Model-based Testing). Because of timing limitations this solution couldn’t be explored in this thesis.

7 Bibliography

[1] G. Keßler, D. Sieben, A. Bhange und E. Börner, „The Software Defined Vehicle – Technical and Organizational Challenges and Opportunities," in *23. Internationales Stuttgarter Symposium*, Stuttgart, 2023.

[2] H. Askaripoor, M. H. Farzaneh und A. Knoll, „E/E Architecture Synthesis: Challenges and Technologies," *MDPI electronics,* Bd. 11, Nr. 518, 2022.

[3] T. Schulze und J.-E. Stavesand, „Hardware-in-the-Loop Test Process for Modern E/E Systems," in *Simulation and Testing for Vehicle Technology, 7th Conference*, Berlin, 2016.

[4] M. RUMEZ, D. GRIMM, R. KRIESTEN und E. SAX, „An Overview of Automotive Service-Oriented Architectures and Implications for Security Countermeasures," *IEEE Access,* Bd. 8, pp. 221852-221870, 2020.

[5] N. Navet, F. Simonot-Lion, R. Zurawski, L. Lavagno, G. Martin und L. Scheffer, Automotive Embedded Systems Handbook, Boca Raton, FL: CRC Press, 2009.

[6] H. Windpassinger, „On the Way to a Software-defined Vehicle," *Springer Professional ATZelectronics worldwide,* Nr. 7-8, pp. 48-51, 2022.

[7] A. Mattausch, J. Schlosser und M. Neukirchner, „E/E Architectures and the Automotive OS," in *23. Internationales Stuttgarter Symposium- Automobil- und Motorentechnik*, Stuttgart, 2023.

[8] W.-H. Kaps, S. Sckuhr und M. Lunt, „AUTOSAR 20th Anniversary: Achievements and Trends on the Example of Basic Software (BSW)," in *23. Internationales Stuttgarter Symposium*, Stuttgart, 2023.

[9] H. A. Oral, „An Effective Modeling Architecture for MIL, HIL and VDIL Testing," *SAE International J. Passeng. Cars - Electron. Electr. Syst.,* Bd. 6, Nr. 1, pp. 34-45, 2013.

[10] P. B. Renaux, R. R. Linhares und D. P. B. Renaux, „Hardware-In-The-Loop Simulation Low-Cost Platform," in *2017 VII Brazilian Symposium on Computing Systems Engineering*, Curitiba, Paraná, Brazil, 2017.

[11] C. Park, S. Chung und H. Lee, „Vehicle-in-the-Loop in Global Coordinates for Advanced Driver Assistance System," *MDPI Applied Sciences,* Bd. 10, Nr. 2645, p. 15, 2020.

[12] T. Hilpert und O. Tetz, „Developing a Fully Automated Prototype for Scaling Automotive Software Tests in the Cloud," in *23. Internationale Stuttgarter Internationale Stuttgarter*, Stuttgart, 2023.

[13] G. Vitale und M. Hollander, „The Software-Defined Vehicle: How to Verify and Validate Software Functions," in *23. Internationales Stuttgarter Symposium*, Stuttgart, 2023.

[14] N. Brayanov und A. Stoynova, „Review of hardware-in-the-loop – a hundred years progress in the pseudo-real testing," *Electrotechnica & Electronica (E+E),* Bd. 54, Nr. 3-4, pp. 70-84, 2019.

[15] ISO, ISO 26262-6 Road vehicles — Functional safety, Switzerland: ISO, 2018.

[16] P. Waeltermann, „WHITE PAPER Hardware-in-the-Loop: The Technology for Testing Electronic Controls in Vehicle Engineering," dSPACE Inc., MI, USA, 2020.

[17] A. Bouscayrol, „Different types of Hardware-In-the-Loop simulation for electric drives," in *2008 IEEE International Symposium on Industrial Electronics*, Cambridge, UK, 2008.

[18] K. NAIK und P. TRIPATHY, Software Testing and Quality Assurance Theory and Practice, Hoboken, New Jersey: Wiley, 2008.

[19] F. Mihalič, M. Truntič und A. Hren, „Hardware-in-the-Loop Simulations: A Historical Overview of Engineering Challenges," *MDPI Electronics,* Bd. 11, Nr. 15, p. 2462, 2022.

[20] E. Dustin, J. Rashka und J. Paul, Automated Software Testing Introduction, Management, and Performance, Castleton, New York: Addison-Wesley, 2008.

[21] N. Englisch, R. Bergelt und W. Hardt, „An AUTOSAR-Specific Static Testing Strategy for Educational Automotive Software Engineering," in *2021 10th Mediterranean Conference on Embedded Computing (MECO)*, Budva, Montenegro, 2021.

[22] N. Englisch, R. Mittag, F. Hänchen, O. Khan, A. Masrur und W. Hardt, „Efficiently Testing AUTOSAR Software Based on an Automatically Generated Knowledge Base," in *Simulation and Testing for Vehicle Technology 7th Conference*, Berlin, 2016.

[23] E. Dustin, T. Garrett und B. Gauf, „Implementing Automated Software Testing," Pearson Education, Inc., Boston, MA, 2009.

[24] N. Kaul, Implementing Automated Software Testing, Burlington, ON, Canada: Arcler Press, 2023.

[25] ISO, „ISO 9126 Information technology - Software product quality," 2000.

[26] J. Zander, P. Mosterman und I. K. Schieferdecker, Model-Based Testing for Embedded Systems, CRC Press, 2017.

[27] VECTOR, „Testing with CANoe," 2009.

[28] VECTOR, „CANoe Product Information," 2023.

[29] VECTOR, „Diagnostics with CAPL," 2023.

[30] VECTOR, Programming with CAPL, Novi, MI, USA: Vector CANtech, 2004.

[31] F. Zhou, S. Li und X. Hou, „Development Method of Simulation and Test System for Vehicle Body CAN Bus Based on CANoe," in *Proceedings of the 7th World Congress on Intelligent Control and Automation*, Chongqing, China, 2008.

[32] BOSCH, CAN Specification 2.0, Stuttgart: Robert Bosch GmbH, 1991.

[33] ISO, „ISO 11989 Road vehicles - Controller area network (CAN) - Part 1: Data link layer and physical signalling," 1999.

[34] BOSCH, „CAN with Flexible Data-Rate Specification," 2012.

[35] Vector Informatik GmbH, „CANoe and CANalyzer as Diagnostic Tools," Vector, 2022.

[36] ISO, ISO 14229 Road vehicles — Unified diagnostic services (UDS) — Specification and requirements, Switzerland: ISO, 2006.

[37] D. S. John, D. V. S. Kumar, P. Ramalingam und M. Murugesan, „Diagnostic feature automation and diagnostic tool for In – Vehicle Infotainment (IVI) system," *International Journal of Pure and Applied Mathematics,* Bd. 119, Nr. 7, pp. 1093-1099, 2018.

[38] D. Ulmer, S. Wittely, K. Hunlichy und W. Rosenstiel, „A Hardware-in-the-Loop Testing Platform Based on a Common Off-The-Shelf Non-Real-Time Simulation PC," in *The Sixth International Conference on Systems ICONS 2011*, St. Maarten, The Netherlands Antilles, 2011.

[39] MathWorks, „WHITE PAPER Hardware-in-the-Loop Testing for Power Electronics Control Design," The MathWorks, Inc, 2019.

[40] Y. Xu, P. Ou und Y. Li, „Design of Vehicle Gateway Automatic Test System Based on CANoe," in *IEEE 2019 Chinese Automation Congress (CAC)*, Hangzhou, China, 2019.

[41] S. Freiberger, M. Albrecht und J. Käufl, „Reverse Engineering Technologies for Remanufacturing of Automotive Systems Communicating via CAN Bus," *Journal of Remanufacturing,* Bd. 1, Nr. 6, 2011.

[42] VECTOR, „Diagnostics via CANoe Gateways," 2022.

Chemnitzer Informatik-Berichte

In der Reihe der Chemnitzer Informatik-Berichte sind folgende Berichte erschienen:

CSR-20-01 Danny Kowerko, Chemnitzer Linux-Tage 2019 - LocalizeIT Workshop, Januar 2020, Chemnitz

CSR-20-02 Robert Manthey, Tom Kretzschmar, Falk Schmidsberger, Hussein Hussein, René Erler, Tobias Schlosser, Frederik Beuth, Marcel Heinz, Thomas Kronfeld, Maximilian Eibl, Marc Ritter, Danny Kowerko, Schlussbericht zum InnoProfile-Transfer Begleitprojekt localizeI, Januar 2020, Chemnitz

CSR-20-03 Jörn Roth, Reda Harradi und Wolfram Hardt, Indoor Lokalisierung auf Basis von Ultra Wideband Modulen zur Emulation von GPS Positionen, Februar 2020, Chemnitz

CSR-20-04 Christian Graf, Reda Harradi, René Schmidt, Wolfram Hardt, Automatisierte Kameraausrichtung für Micro Air Vehicle basierte Inspektion, März 2020, Chemnitz

CSR-20-05 Julius Lochbaum, René Bergelt, Time Pech, Wolfram Hardt, Erzeugung von Testdaten für automatisiertes Fahren auf Basis eines Open Source Fahrsimulators, März 2020, Chemnitz

CSR-20-06 Narankhuu Natsagdorj, Uranchimeg Tudevdagva, Jiantao Zhou, Logical Structure of Structure Oriented Evaluation for E-Learning, April 2020, Chemnitz

CSR-20-07 Batbayar Battseren, Reda Harradi, Fatih Kilic, Wolfram Hardt, Automated Power Line Inspection, September 2020, Chemnitz

CSR-21-01 Marco Stephan, Batbayar Battseren, Wolfram Hardt, UAV Flight using a Monocular Camera, März 2021, Chemnitz

CSR-21-02 Hasan Aljzaere, Owes Khan, Wolfram Hardt, Adaptive User Interface for Automotive Demonstrator, Juli 2021, Chemnitz

CSR-21-03 Chibundu Ogbonnia, René Bergelt, Wolfram Hardt, Embedded System Optimization of Radar Post-processing in an ARM CPU Core, Dezember 2021, Chemnitz

CSR-21-04 Julius Lochbaum, René Bergelt, Wolfram Hardt, Entwicklung und Bewertung von Algorithmen zur Umfeldmodellierung mithilfe von Radarsensoren im Automotive Umfeld, Dezember 2021, Chemnitz

Chemnitzer Informatik-Berichte

CSR-22-01 Henrik Zant, Reda Harradi, Wolfram Hardt, Expert System-based Embedded Software Module and Ruleset for Adaptive Flight Missions, September 2022, Chemnitz

CSR-23-01 Stephan Lede, René Schmidt, Wolfram Hardt, Analyse des Ressourcenverbrauchs von Deep Learning Methoden zur Einschlagslokalisierung auf eingebetteten Systemen, Januar 2023, Chemnitz

CSR-23-02 André Böhle, René Schmidt, Wolfram Hardt, Schnittstelle zur Datenakquise von Daten des Lernmanagementsystems unter Berücksichtigung bestehender Datenschutzrichtlinien, Januar 2023, Chemnitz

CSR-23-03 Falk Zaumseil, Sabrina, Bräuer, Thomas L. Milani, Guido Brunnett, Gender Dissimilarities in Body Gait Kinematics at Different Speeds, März 2023, Chemnitz

CSR-23-04 Tom Uhlmann, Sabrina Bräuer, Falk Zaumseil, Guido Brunnett, A Novel Inexpensive Camera-based Photoelectric Barrier System for Accurate Flying Sprint Time Measurement, März 2023, Chemnitz

CSR-23-05 Samer Salamah, Guido Brunnett, Sabrina Bräuer, Tom Uhlmann, Oliver Rehren, Katharina Jahn, Thomas L. Milani, Güunter Daniel Rey, NaturalWalk: An Anatomy-based Synthesizer for Human Walking Motions, März 2023, Chemnitz

CSR-24-01 Seyhmus Akaslan, Ariane Heller, Wolfram Hardt, Hardware-Supported Test Environment Analysis for CAN Message Communication, Juni 2024, Chemnitz

Chemnitzer Informatik-Berichte

ISSN 0947-5125

Herausgeber: Fakultät für Informatik, TU Chemnitz
Straße der Nationen 62, D-09111 Chemnitz