Hindawi
Mobile Information Systems
Volume 2017, Article ID 4395070, 13 pages
https://doi.org/10.1155/2017/4395070

Research Article

Collecting Big Data from Automotive ECUs beyond the CAN Bandwidth for Fault Visualization

Jeong-Woo Lee, Ki-Yong Choi, and Jung-Won Lee

Department of Electrical and Computer Engineering, Ajou University, Suwon, Republic of Korea

Correspondence should be addressed to Jung-Won Lee; jungwony@ajou.ac.kr

Received 9 December 2016; Accepted 7 February 2017; Published 27 February 2017

Academic Editor: Jeongyeup Paek

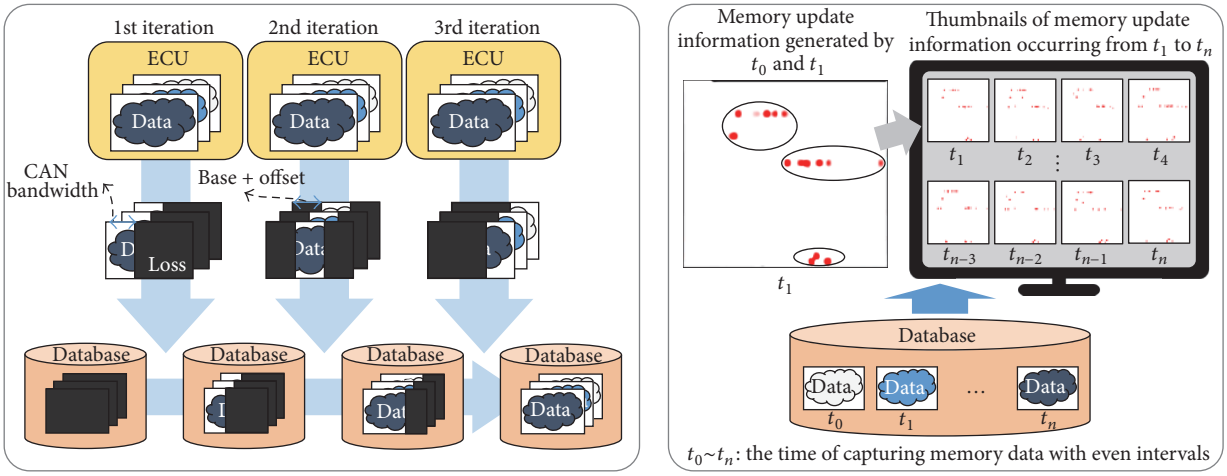
A hardware-in-the-loop (HiL) test is performed to verify the software functions mounted on automotive electronic control units (ECUs). However, the characteristics of HiL test limit the usage of common debugging techniques. Meanwhile, the logs of how the program uses memory can be utilized as debugging information collected by the controller area network (CAN). However, when the 32 KB memory is observed with 10 ms period, about 96% of the data on each cycle is lost, since the CAN only can transfer 1.25 KB of data at each cycle. Therefore, to overcome the above limitations, in this study, the memory is divided into multiple regions to transmit generated data via CAN. Next, the simulation is repeated for the each divided regions to obtain the different areas in each simulation. The collected data can be visualized as update information in each cycle and the cumulative number of updates. Through the proposed method, the ECU memory information during the HiL test was successfully collected using the CAN; the transmission is completed without any loss of data. In addition, the data was visualized in images containing the update information of the memory. These images contribute to shortening the debugging time for developers and testers.

1. Introduction

The software installed in an automotive electronic control unit (ECU) has increased significantly in size and complexity. This automotive software is responsible for features directly connected to the user's safety, such as the advanced driver assistance system (ADAS) or the adaptive cruise control (ACC). Automotive software is developed using the V-model development process, with corresponding verifying methods in each stage. The verification method is separated into model, software, processor, and hardware-in-the-loop (MiL, SiL, PiL, and HiL) tests, depending on the integration level of each stage. Among them, the HiL test is performed to check whether the functions meet the requirements through the interaction between the software mounted on the completely developed ECU and the peripheral hardware [1].

In the HiL test, the HiL simulator provides input to the target hardware, which is treated as a black box, and observes the corresponding results [2]. When the faults are detected in the test results, the developers should proceed to the debugging. Mostly, debugging the hardware is proceeded by using the interface such as joint test action group (JTAG) to monitor the internal behavior of the target. But, there are some limitations to obtaining the debugging information in the HiL test environment. First, when performing the test with the completed ECU form, it is hard to modify the hardware when debug interfaces, such as JTAG and the background debug mode (BDM), do not exist or are not exposed. Further, stopping the system arbitrarily, for example, to obtain debugging information, is limited. To interrupt the HiL test in progress at any time, both the ECU and the HiL simulator must be stopped at the same time. However, the HiL simulator is configured independently from the ECU; thus, there is a limit to simultaneously operating it at the system-clock level [3].

Meanwhile, the memory information collected periodically during the program execution can be utilized as debugging information. Embedded systems, such as the ECU, execute the actions defined in a program by loading data from memory and storing the updated data. These operations are performed using memory-interface instructions defined in the program [4, 5]. In other words, observing the logs hat uses the memory can be useful for understanding the program operation. However, since the information generated



(a) Data collection (b) Data visualization

FIGURE 1: Collecting and visualizing memory data beyond the CAN bandwidth.

regularly in the memory is large, techniques for transmitting the data are required. For example, over 3 MB of data is generated per second when 32 KB of data is generated every 10 ms. To provide the data generated during the HiL test to the developer, the following challenges should be considered.

First, methods are needed for overcoming the network bandwidth limitation. The FlexRay, controller area network (CAN), and local interconnect network (LIN) can be utilized to transfer the large amount of memory data generated inside the ECU while executing the HiL test to the outside [6]. Among them, the FlexRay, which is not commonly used in the ECU, is unsuitable for general use. The LIN cannot be used for large data transmissions because of its low bandwidth: about 20 Kbps [7]. Therefore, the CAN protocol is appropriate for collecting massive memory data [8]. However, since the CAN's maximum transfer rate is 1 Mbps, the amount of data that can be transmitted in the above environment is 1.25 KB per 10 ms; that is, 96% of the data is lost. An additional technique is required to overcome this limitation.

Attempts have been made to overcome the limitations when transferring large amounts of data using CAN by modifying the CAN protocol or compressing and fragmenting data. However, existing methods are improper for the large data transmissions required in the HiL test environment. First of all, the technique for modifying the CAN protocol requires a revision in the communication driver to identify the modified protocol [9–11]. Secondly, the technique to compress the data can interrupt the existing operation because it inflicts an extra load on the processor. Moreover, when the compression ratio does not satisfy the requirement, the amount of uncompressed data can be lost or communication can be delayed [12–16]. Lastly, in the technique to fragment the data, since the ECU has a limited buffer size, the previously generated, unsaved data can be lost [17].

Meanwhile, by using the transfer technique in Figure 1(a), a large amount of memory data can be transmitted outside of the ECU using CAN by repeating the simulation. Due to the automotive software characteristics that take charge of the user's safety and require strict real-time operation, the variables for a specific function always operate deterministically [18]. Using this characteristic, the data generated in each separate region is sent by repeating the simulation. The size of each memory region is determined by considering the memory size, the communication environment, and the ECU operation period. The transferred data is stored in the database along with its order information. Using this technique, large amounts of data can be sent outside without requiring hardware modifications or interfering with existing operations.

When providing the memory data collected from the ECU to the developer and tester, it is more effective to provide a visualized image than text [19, 20]. When the collected data is provided in text, they must check the values collected from the 32-KB memory generated every 10 ms. However, by providing an image, the developer can observe the updates of each memory address on a 256 × 128-entry image. Based on the information in the image, the developer can intuitively select a focus area. The collected memory data can be visualized as shown in Figure 1(b). The data collected periodically during the simulation, from t_0 to t_n, is stored in a database. The memory-update information in t_1 can be acquired by comparing it with the data generated in t_0. The memory-update information in t_2–t_n can be similarly acquired. By visualizing the acquired memory-update information, it is possible to monitor the operations in the memory via images during the program execution. In addition, it is possible to shorten the debugging time by eliminating unnecessary information that is not related to the operations.

Therefore, in this paper, we propose techniques for collecting the large amount of memory data generated periodically in the ECU during the HiL test and visualizing the text data effectively. First, we segment the memory area into a size that can be transferred per unit time, considering the size of the memory and the communication environment. Then, we repeat the simulation for each segmented area to collect the data generated there. Lastly, we acquire the memory-update

information from the collected data to visualize the update information in each unit time and the cumulative update frequency in each address. By using our proposed method, massive data generated from the ECU can be transmitted to the external database. The collected data can be useful information for the developers to understand the defective behavior of the target.

The rest of this paper is organized as follows. Section 2 explains the limitations to acquiring debugging information in the general HiL test environment and existing methods to overcome the CAN bandwidth. In Section 3, methods to collect memory data using the data-cascading method and the visualization of the collected data are explained. In Section 4, modules to process the proposed methods are described. Section 5 presents the implementation of the methods and the verification of the collected data. The conclusion is presented in Section 6.

2. Related Work

In this chapter, the automotive HiL test, the limitations that occur when acquiring debug information in an environment with restricted hardware modifications, and limitations in utilizing system resources are described. Then, to overcome the transmission limit of the CAN protocol, a conventional method for transferring a large volume of data and its limitations for applying them in the HiL test environment are explained.

2.1. Hardware-in-the-Loop (HiL) Testing. Automotive software is developed in accordance with the V-model development process and has a variety of verification methods, depending on the level of system integration at each stage. Among them, the HiL test takes charge of testing the interaction between the software mounted on the fully developed ECU and the peripheral hardware. The reasons for performing the HiL test are as follows. First, by building a simulation environment, it is possible to reduce the costs incurred in initially establishing and then maintaining a real-car environment [21]. Furthermore, improving the test coverage, increasing the test reproducibility, and shortening the test time can be accomplished through test automation [22].

In general, the HiL test environment consists of a host PC, the HiL simulator, and the ECU. The host PC transfers the test scripts to the HiL simulator; they include use scenarios for running the completely developed software in an automobile according to the testing purpose. The HiL simulator receives the test script from the host PC via a high-speed network such as Ethernet and delivers the test inputs stated in the script to the ECU via I/O or communication interfaces. Lastly, the system under test is composed of the ECU and the peripheral hardware. The ECU in the system receives the inputs based on the test script, performs the operation according to the inputs, and delivers the output to the HiL simulator to determine the presence of faults.

However, obtaining debug information on the occurred faults has the following limitations. Firstly, it is impossible to use commonly used hardware debuggers, such as JTAG or BDM, when the interfaces are either absent or not exposed to the outside [22]. As the testing process proceeds to subject the completely developed ECU, modifications for appending a debugging interface are prohibited. When debugging ports are available to use, the test cannot be stopped arbitrarily to collect data because of the test script which includes the use scenario. To suspend the HiL test, not only the ECU, but also the HiL simulator must be stopped at the exact same time. However, the HiL simulator, which differs from an In-Circuit Emulator (ICE) or a software debugger, is configured independently of the ECU; thus, it is limited to simultaneously controlling both systems at the system-clock level. In addition, utilizing the source code to detect the behavior causing the fault is restricted, because the testing target is treated as a black box.

Meanwhile, embedded systems such as ECUs use memory-interface commands, such as MOVE, LOAD, or STORE, to acquire data stored in the memory for processing the operation and to store the data into memory, for example, RAM. In other words, since RAM is utilized to store the data changed during the operation stated in the test, periodically monitoring the data updates in memory can be used to trace the behavior of the program processed inside the ECU. Therefore, by monitoring the memory of the ECU during the program execution in the HiL test environment, it is possible to acquire debugging information related to the faults [4, 5].

2.2. Transferring Large Automotive Data Using the CAN Bus. The CAN (Controller Area Network) is a protocol that is commonly used in all ECUs, with a transmission rate from 25 Kbps up to 1 Mbps. However, such a transfer rate is optimized to share small but critical values, such as RPM (revolutions per minute) or temperature values [23]. Thus, in addition to the state information generated while performing conventional operations, another approach is needed to overcome the transmission limitations when transferring to the outside the large amount of data that is also generated during the operation, for example, additional information for debugging. The following techniques have been proposed for using the CAN to transfer big data.

Firstly, methods for improving the CAN protocol's bandwidth have been proposed. The CAN+ protocol increases the amount of data that can be sent in a message by setting the high-transfer rate in the gray zone existing between regions for synchronizing and sampling the data [9, 10]. FastCAN overcomes the transfer limits by adding an extra bus to the network, which originally consisted of a single bus [11]. Further, it mediates messages via a new medium-access technique, allowing each node to efficiently access the bus. Although both approaches improve the transfer rate by 16 times or more, they cannot be utilized to transfer a large amount of data to the outside in the HiL test environment. When any field in the protocol is modified, except the data field, an overall conversion is required to allow the communication driver to recognize the data in that area. However, since the HiL test proceeds on the assumption that the manufacturing ECU is complete, hardware modifications are not appropriate.

Secondly, reducing the large volume of data by using a compression algorithm has been proposed. Wu et al. presented an algorithm which performs the delta compression to reduce the amount of data when it exceeds 5 bits [12]. Additionally, they suggested another approach to enhance the data compression ratio by rearranging different signals according to their characteristics [13]. In Miucic et al.' research [14], additional control bits are used to compare the original data to the compressed data in order to send the smaller one. To enhance the algorithm proposed by Miucic, Kelkar suggested the method to diversify the parameters by utilizing the fields that represent the compression information [15]. In advance, the computation is reduced by positioning the compression information in front of the compressed data unlike the above [16]. However, applying such compression algorithms has the following restrictions in the HiL test environment. First, when the compression ratio does not satisfy the communication-bandwidth requirement, the surplus data can be lost. Furthermore, the existing test behavior can be interrupted since the operations for compressing data can be an additional burden on the processor.

Finally, a technique for dividing the data into a size that can be stored in a message has been proposed. The data is divided into segments that can be stored in a message, including the sequence information; then, each segment is transmitted [17]. Using the eight-byte data field in the CAN protocol, the data is stored in six bytes with two bytes of sequence information. In addition, the reserved bits in the control field are used to indicate fragmentation. However, this is not applicable to the HiL test, where a large amount of data is generated periodically inside the ECU. Embedded systems, such as ECUs, have limited storage buffers. Therefore, when monitoring the ECU while the HiL test is in progress and a large amount of memory data is repeatedly generated, there is a possibility that the data will overflow the storage buffer and be lost. Furthermore, when using reserved bits for sequence information, the communication driver must be changed to interpret the modified message, and this cannot be applied to the ECU in the HiL test.

3. Collecting Memory Data for Fault Visualization

This section explains the method for collecting large amounts of memory data periodically by using a data-cascading method in the ECU while executing the HiL test and the algorithm to calculate the memory-update information using the collected data. Moreover, methods for a developer to visualize the update information are suggested.

3.1. Data-Cascading Method for Collecting Extensive Memory Data. To transfer the large amount of memory data generated by the ECU in the HiL test, limitations must be overcome, such as restrictions in utilizing system resources or modifying the hardware. In the data-cascading method, extensive data was successfully transmitted by running a simulation repeatedly with the same test scenario [24]. Figure 2 illustrates the process of collecting the memory data generated inside the ECU using the above method. For example, considering the operation cycle of the system's main task as 10 ms and the size of the memory to monitor periodically as ①②③, the following process is executed to divide an area into the size of data that can be transferred in each cycle.

First, the number of messages that can be sent in one cycle is calculated by dividing the operation cycle into the time required to send one message successfully. In the example, three messages can be sent in one operation cycle (①-1, ①-2, and ①-3). By considering the bus occupancy of the existing operation, it is possible to calculate the number of messages that can be sent without affecting the execution. The size of the area that can be transferred per operation cycle is calculated by multiplying the size of the storable data in a message, excluding the sequence information.

In Figure 2, the area to monitor is divided into three regions, and the amount of data that can be sent in one cycle is ①. Therefore, to collect the data generated in all of the regions, the simulation must be repeated three times. In the first simulation, the data corresponding to region ① is sent, and in the second simulation, the data corresponding to region ② is sent. This process is repeated until all the data regions are sent to the database.

To collect the data periodically generated in the monitored area, as in Figure 2, the number of divided areas and the simulation repetitions can be calculated using the following information:

(i) P_{task}: operation cycle of the system's main task;

(ii) P_{msg}: minimum time to transfer a message;

(iii) N: number of messages that can be sent in P_{task} ($P_{\text{task}}/P_{\text{msg}}$);

(iv) D_{req}: size of data generated in every P_{task};

(v) D_{msg}: size of data stored in a message, excluding the order information;

(vi) OCC: occupancy of the communication bus for the existing operation.

By considering the existing bus occupancy, the number of messages to send in each cycle is calculated as $N \times D_{\text{msg}} \times (1 - \text{OCC})$. Finally, the number of area segments is calculated as dividing D_{req} by $N \times D_{\text{msg}} \times (1 - \text{OCC})$, as in

$$\#\ \text{of segments} = \frac{D_{\text{req}}}{N \times D_{\text{msg}} \times (1 - \text{OCC})}. \quad (1)$$

Meanwhile, the size of the data field in the CAN data frame is eight bytes. To utilize it efficiently, it is necessary to define the proper protocols. Figure 3 shows three protocol types; their detailed explanations follow.

(i) Data-Request Protocol (RP). The data collector in the host PC generates a data-request packet (RP packet) to send the transfer environment and the area to request to the transfer agent in the ECU. The information field is set to 1 to indicate that it is an RP packet. In the rest of the fields, the total number

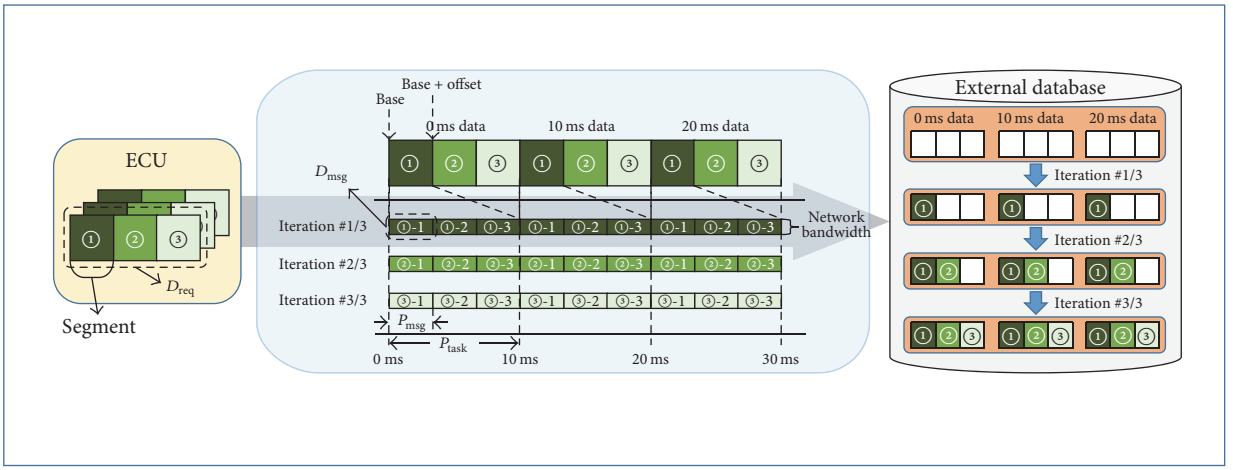


Figure 2: Overall process of moving memory data from the ECU to an external database using a data-cascading algorithm.

Protocol type	Bytes							
	0		1	2	3	4	5	6	7
	0	1–7							
Data request (RP)	Info.	Reserved		# of frames		Start address		End address	
Cascading information (CIP)	Info.	Offset		Frame index		Reserved			
Data transfer (DP)	Info.	Base address		Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

Figure 3: Protocols defined in the data-cascading algorithm.

of frames and the start and end addresses to monitor are included.

(ii) Cascading-Information Protocol (CIP). The transfer agent generates the cascading-information packet (CIP packet) to transfer the sequence information of the data currently transmitted. The information field is set to 1 to indicate that it is the CIP packet. The rest of the fields include the sequence information among the divided regions (offset) and the frame index of the frame to which the currently transmitted data belongs.

(iii) Data-Transfer Protocol (DP). The transfer agent generates a data-transfer packet (DP packet) to transfer the data with the sequence information in the divided regions. The information field is set to 0 to indicate that it is the DP packet. In the rest of the fields, six bytes of successive data is stored, and their starting address is stored in the base address field.

3.2. Computing the Memory-Update Frequency for Fault Visualization. Using the suggested method, the memory data generated in the ECU is collected to acquire the memory-update information. The memory data (D_{req}) is collected in every P_{task} during the execution time (E). Equations (2) and (3) represent the address range and the frame index. The frame is defined in (4), which represents the set of data generated in each cycle.

$$\text{Memory Address Set}\,(A) \equiv \left\{a \mid D_{req,start} \le a < D_{req,end};\ a \text{ is an address}\right\} \tag{2}$$

$$\text{Frame Index Set}\,(I) \equiv \left\{k \mid 0 \le k < \frac{E}{P_{task}};\ k \text{ is a frame index}\right\} \tag{3}$$

$$\text{Frame}\,(F_{a,k}) \equiv \{v_{a,k} \mid \text{Value stored in address } a \text{ generated at time } k \times P_{task};\ a \in A,\ k \in I\}. \tag{4}$$

The memory-update information is calculated as in (5). By comparing two consecutive frames, the equation assigns 1 when the data has changed and 0 when the data has not changed, or it is the 0th frame.

$$\text{Memory Update}\,(U_{a,k}) = \begin{cases} 0 & \text{if } k = 0 \\ 0 & \text{if } v_{a,k-1} = v_{a,k} \\ 1 & \text{if } v_{a,k-1} \neq v_{a,k} \end{cases} \quad (a \in A, k \in I). \tag{5}$$

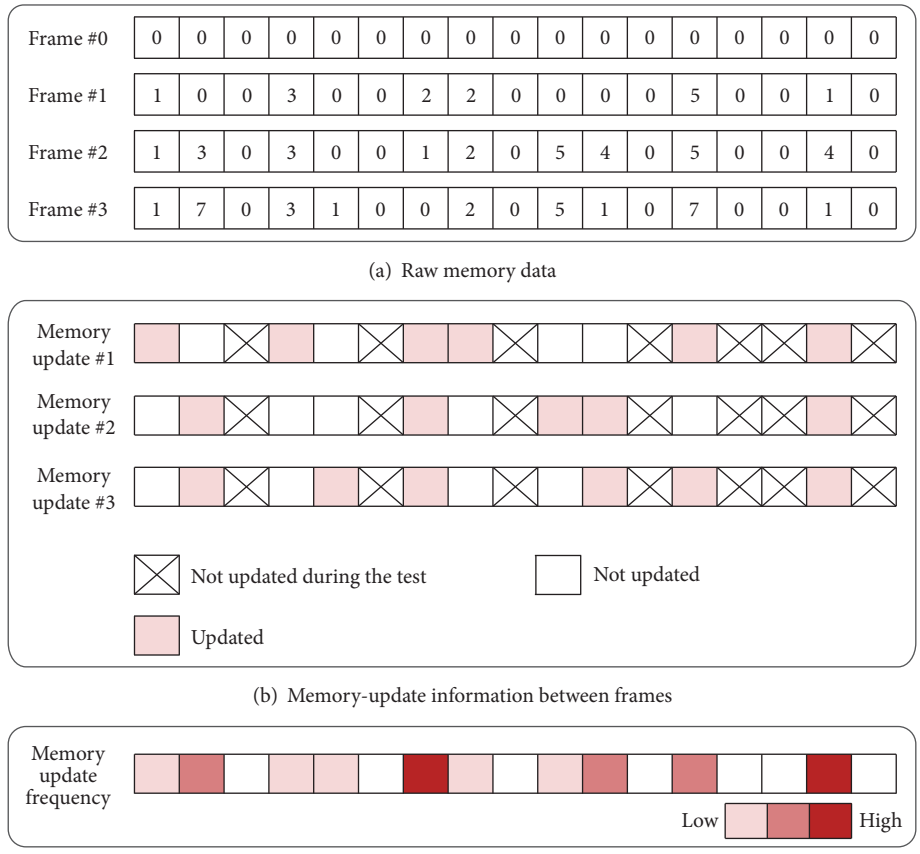


Figure 4: Converting raw memory data into images of the frames containing the update information and an image containing the total memory-update frequency.

Using the equation above, it is possible to obtain the memory-update information of the kth frame. With this information, it is possible to calculate the cumulative update count for each memory address generated during execution, as expressed in

$$\text{Memory Update Frequency}\left(\text{UF}_{A,I}\right) = \sum_{k \in I} U_{a,k} \quad (a \in A). \tag{6}$$

3.3. Visualizing the Memory-Update Frequency. The collected memory data is stored in frames, as shown in Figure 4(a). Figure 4(a) represents $F_{A,0}$–$F_{A,3}$, which is the raw data stored in the database. By processing the raw data, $U_{A,1}$–$U_{A,3}$, $UF_{A,I}$ can be obtained for visualization as Figures 4(b) and 4(c).

Visualizing $U_{A,k}$ in Each Time k (Visualizing the Update Information in Each Frame). $U_{A,k}$ is visualized with the data acquired in (5). Except the 0th frame, the images are generated as the number of frames, and each image contains the information as $U_{A,1}$–$U_{A,3}$ in Figure 4(b). Each image shows whether an update occurred in each address. An address where no update occurred or randomly occurred can be checked differently to reduce the regions to monitor.

Visualizing $UF_{A,I}$ (Visualizing the Total Update Frequency). $UF_{A,I}$ is visualized in a single image. As shown in Figure 4(c), the information is provided with an image to grasp the approximate update frequency by using different colors to represent the number of updates. With this image, it is possible to reduce the time spent debugging by preferentially selecting the memory regions to monitor.

4. Design of Modified Architecture in the HiL Test Environment

To collect and visualize the data generated from the ECU while processing the HiL test, the following modules have been designed. Figure 5 depicts the modified architecture from the general HiL test environment. In the host PC, the data collector and visualizer are installed. In the ECU, a transfer agent is installed to perform the functions. The transfer agent in the ECU and the data collector in the host PC are connected to each other, using the CAN to exchange packets. The visualizer generates the images with the data from the database. A detailed explanation of each module follows.

4.1. Data Collector. The data collector is installed in the host PC and provides a UI (user interface) to receive the

Input:
CIP - Cascading information packet from transfer agent
DP - Data transfer packet from transfer agent
Output:
$v_{a,k} \sim v_{a+5,k}$ - Values generated at kth frame in address a to $a + 5$
(1) **procedure** ReceiveDataSegments
(2) **begin**
(3) $k \leftarrow$ CIP.*Frame*
(4) *offset* $\leftarrow$ CIP.*Offset*
(5) **if** DP received **then**
(6) $a \leftarrow$ DP.*Base*
(7) *data*[ ] $\leftarrow$ DP.*Data*[ ]
(8) Stores $v_{\text{offset}+a,k} \sim v_{\text{offset}+a+5,k}$ into the database
(9) **end if**
(10) **end**
(11) **end procedure**

Algorithm 1: Algorithm of the data collector collecting the data.

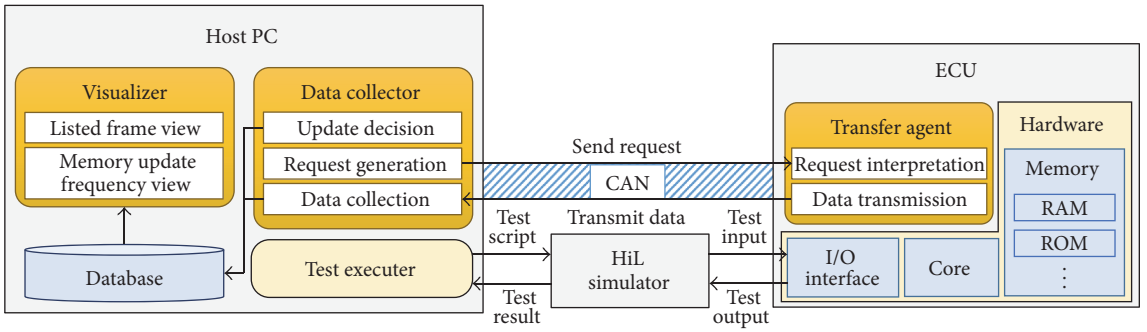


Figure 5: Modified architecture of the HiL test environment.

information from the user about the test environment and the memory area to monitor. In addition, it provides the following functions. First, it generates an RP packet based on the information requested by the user and sends the packet to the transfer agent. Second, it receives the DP packet and CIP packet from the transfer agent, parses the packets, and stores the data in the database. Finally, it acquires the memory-update information by comparing the data received and the data previously stored. The detailed algorithm to store the data from received packets is depicted in Algorithm 1.

The data collector receives the CIP packet and DP packet from the transfer agent and parses them to acquire the data and sequence information to store in the database. Firstly, the offset and the frame index are acquired by analyzing the received CIP packet (lines (3)-(4)). After obtaining the sequence information, the data acquired from the DP packet is stored in the database. From the DP packet, six consecutive bytes of data and their starting address can be acquired (lines (6)–(8)).

Algorithm 2 shows an algorithm for calculating the update information by using the received data. First of all, when the data collector receives the CIP packet, it saves the value of the frame index into k by paring the packet (line (4)). Then, receiving the following DP packets, it stores the consecutive six bytes of data and their starting address into a (line (6)). If the value of k equals 0, it assigns the value 0 from $U_{a,0}$ to $U_{a+5,0}$ (lines (7)-(8)). If the value of k is greater than 0, it compares the received data to the data in the buffer to check the difference. When there is a difference, it assigns 1 to the area where the update occurred (lines (11)–(17)).

4.2. Transfer Agent. The transfer agent is implemented as independent software in the ECU; it exchanges packets with the data collector in the host PC by the connection to the CAN. It receives the RP packet from the data collector to obtain the range of memory to monitor and the total frame count. After obtaining the information, it captures the data according to the request, produces the DP packet, and sends it to the data collector. In addition, it sends a CIP packet to transmit the sequence information of the transmitting data. The algorithm for the above functions is depicted in Algorithm 3.

The transfer agent receives the RP packet from the data collector and the environment information, such as P_{task}, P_{msg}, D_{req}, D_{msg}, and OCC. With the information, the transfer agent operates as follows. Firstly, it obtains the value of the frame index and the start and end addresses of the memory area to monitor from the DP packet (lines (3)–(5)). Then, it divides the memory according to the environment information. The number of regions to divide

Input:
CIP - Cascading information packet from transfer agent
DP - Data transfer packet from transfer agent
Output:
$U_{a,k} \sim U_{a+5,k}$ - The memory-update occurrence from address a to $a + 5$ at kth frame
(1) **procedure** UpdateDecision
(2) **begin**
(3) buffer[ ]: Temporary buffer for storing the data generated at k-1th frame from address a to $a + 5$
(4) $k \leftarrow$ CIP.*Frame*
(5) **while** Receiving data transfer packet **do**
(6) $a \leftarrow$ DP.*Base*
(7) **if** $k == 0$ **then**
(8) $U_{a,k} \sim U_{a+5,k} \leftarrow 0$
(9) **else**
(10) **for** $i = 0$ to 5 **do**
(11) **if** buffer[i] == DP.*Data*[i]**then**
(12) $U_{a+i,k} \leftarrow 0$
(13) **else**
(14) $U_{a+i,k} \leftarrow 1$
(15) **end if**
(16) **end for**
(17) **end if**
(18) buffer[ ] $\leftarrow$ DP.*Data*[ ]
(19) **end while**
(20) **return** $U_{a,k} \sim U_{a+5,k}$
(21) **end**
(22) **end procedure**

Algorithm 2: Algorithm of the data collector operating the update decision.

can be calculated as shown in Algorithm 4, which requires the values of P_{task}, P_{msg}, D_{req}, D_{msg}, and OCC (Algorithm 4, lines (3)-(4)). When the calculation finishes, it generates the CIP packet using the current frame index and the offset value and the DP packets included in the above sequence information (lines (10)–(13)). The packets are sent to the data collector; this process is repeated while increasing the offset and frame index.

4.3. Visualizer. The visualizer generates images to display the data stored in the database. Two types of images are generated in the visualizer. First, it generates a list of images containing the updated information in each address in each frame. It uses $U_{A,k}$, calculated using (5), and generates the images for the number of frames to check the update information in every cycle. Areas where no updates occurred or areas unrelated to the operation can be filtered out. Second, it creates an image containing the total memory-information count in each address. It generates one image using $UF_{A,I}$, calculated using (6). The image displays the update frequency in different colors to provide an intuitive understanding.

5. Experimental Results

5.1. Memory Data Acquisition. For the implementation, we appended the transfer agent, data collector, and the visualizer modules into the HiL test environment, as shown in Figure 5. The transfer agent and the data collector are installed in the ECU and the host PC to operate the algorithms in Section 4. The environmental information for the dividing memory regions is as follows:

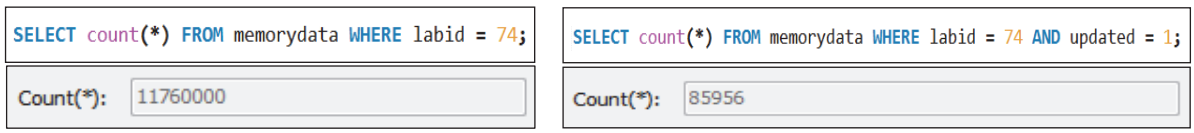
(i) P_{task}: 10 ms;
(ii) P_{msg}: 300 μs;
(iii) $N = P_{\text{task}}/P_{\text{msg}} = (10 \times 10^{-3})/(300 \times 10^{-6}) = 33.33 \approx 33$;
(iv) E: 4 s (# of frames = 4 s/10 ms = 400);
(v) OCC: 55%;
(vi) D_{msg}: 6 bytes;
(vii) $D_{\text{req,start}}$: 0;
(viii) $D_{\text{req,end}}$: 29,400;
(ix) Lab ID: 74.

In the environment above, D_{req} is 29,400 bytes, which means that the amount of data generated in every P_{task}, which is 10 ms, is 29,400 bytes. Since CAN's maximal bandwidth is about 1 Mbps, while the data generated is about 3 MB per second, this causes a loss of about 96% of the data in each cycle. The number of memory regions to divide was calculated as 330, using (7). Therefore, we repeated the

Input:
RP - Data request packet from data collector
P_{task} - Operation cycle of system main task
P_{msg} - The time required for sending one CAN message
D_{msg} - The size of data that can be stored in one CAN message
OCC - CAN bus occupancy of the existing operation
Output:
CIP - Cascading information packet to data collector
DP - Data transfer packet to data collector

```
(1)  procedure SendDataSegments
(2)  begin
(3)      F ← RP.Frame
(4)      a_start ← RP.StartAddress
(5)      a_end ← RP.EndAddress
(6)      x, y ← 0
(7)      D_req ← a_end − a_start
(8)      Seg ← CalculateSegments(P_task, P_msg, D_req, D_msg, OCC)
(9)      while y < Seg do
(10)         while x < F do
(11)             CIP.Frame ← x
(12)             CIP.Offset ← y
(13)             DP.Base ← the address corresponding to offset y
(14)             DP .data[ ] ← 6 bytes of successive data starting from
                              the address of DP.Base
(15)             Send CIP and DP to the data collector
(16)             x ← x + 1
(17)         end while
(18)         x ← 0, y ← y + 1
(19)     end while
(20) end
(21) end procedure
```

Algorithm 3: Algorithm of the transfer agent.



(a) Amount of data collected during the test

(b) Number of updates occurring during the test

Figure 6: Data collection results and the total memory-update count from the HiL test.

simulation 330 times to collect the data generated in the entire region. A and I can be obtained using E, P_{task}, $D_{req,start}$, and $D_{req,end}$, in (8) and (9).

$$\#\ \text{of segments} = \frac{D_{req,end} - D_{req,start}}{N \times D_{msg} \times (1 - OCC)} = \frac{29,400 - 0}{33 \times 6 \times (1 - 0.55)} \approx 330 \tag{7}$$

$$A = \{a \mid 0 \le a < 29{,}400;\ a \text{ is an address}\} \tag{8}$$

$$I = \left\{k \mid 0 \le k < \frac{4}{10 \times 10^{-3}} = 400;\ k \text{ is a frame index}\right\}. \tag{9}$$

In the results shown in Figure 6(a), 400 frames of data, about 11,760,400 bytes, were collected successfully. About 85,956 updates occurred during the program execution; they were calculated using (5).

To verify the collected data, we used the Trace 32 hardware debugger [25]. Trace 32 is a tool that controls and monitors the target through a standard debug interface, such as JTAG or BDM, equipped inside the processor. It is possible to collect the variables, registers, and OS information by suspending the program during execution. However, there is a limit to collecting memory data during the HiL test because this process must stop the execution during the simulation. Therefore, to utilize Trace 32 for verification, we stopped the execution randomly and captured the data stored in the stack, which is a part of memory. We compared the captured stack with the data collected by the proposed method.

Input:

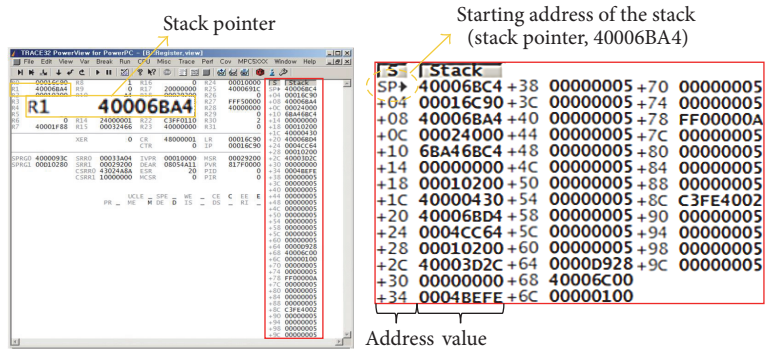
P_{task} - Operation cycle of system main task
P_{msg} - The time required for sending one CAN message
D_{req} - The size of memory to monitor
D_{msg} - The size of data can be stored in one CAN message
OCC - CAN bus occupancy of the existing operation

Output:

Seg - the number of segments to slice

(1) **function** CalculatingSegments(P_{task}, P_{msg}, D_{req}, D_{msg}, OCC)
(2) **begin**
(3) $N \leftarrow D_{msg}/P_{task}$
(4) $Seg \leftarrow D_{req}/(N \times \text{OCC} \times D_{msg})$
(5) **return** *Seg*
(6) **end**
(7) **end function**

Algorithm 4: Function to calculate the number of segments.



(a) Data captured by using Trace 32

Stack pointer

Address	Value	Address	Value	Address	Value
40006BA4	40006BC4	40006BDC	00000005	40006C14	00000005
40006BA8	00016C90	40006BE0	00000005	40006C18	00000005
40006BAC	40006BA4	40006BE4	00000005	40006C1C	FF00000A
40006BB0	00024000	40006BE8	00000005	40006C20	00000005
40006BB4	6BA46BC4	40006BEC	00000005	40006C24	00000005
40006BB8	00000000	40006BF0	00000005	40006C28	00000005
40006BBC	00010200	40006BF4	00000005	40006C2C	00000005
40006BC0	40000430	40006BF8	00000005	40006C30	C3FE4002
40006BC4	40006BD4	40006BFC	00000005	40006C34	00000005
40006BC8	0004CC64	40006C00	00000005	40006C38	00000005
40006BCC	00010200	40006C04	00000005	40006C3C	00000005
40006BD0	40003D2C	40006C08	0000D928	40006C40	00000005
40006BD4	00000000	40006C0C	40006C00		
40006BD8	0004BEFE	40006C10	00000100		

(b) Data collected by using proposed algorithm

Figure 7: Captured data in a stack randomly using Trace 32 and the data collected using proposed algorithm.

Figure 7(a) is the stack and register view; it displays the stack and the stack pointer captured at any time during the execution by Trace 32. On the left side of Figure 7(a), register R1 stores the value of the stack pointer, which is the starting address of the stack, and the right side of Figure 7(a) shows an image of the stack. Starting from address 40006BA4, data is stored on the stack every four bytes. Figure 7(b) is the set of data starting from 40006BA4 collected by the proposed method. We compared the data captured by Trace32 and the data collected by our method and found that they were identical. This means that the data collected by our method can be reliably used.

5.2. Application. To visualize the data collected by the method proposed in this paper, we used the MFC (Microsoft Foundation Class) framework and the VTK (Visualization

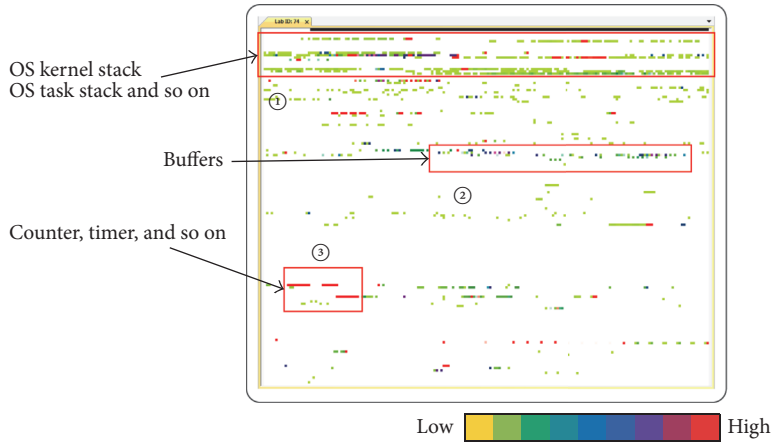


Figure 8: Image of the total memory-update frequency.

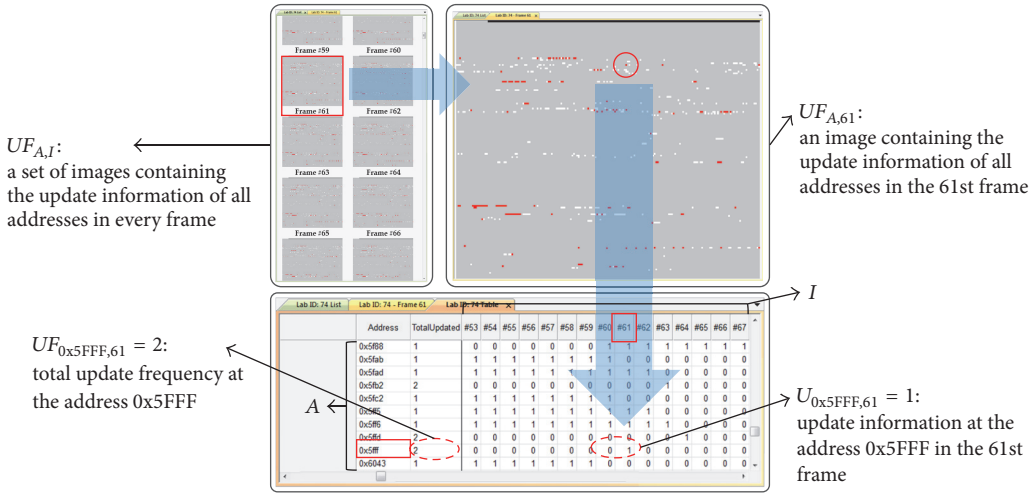


Figure 9: Images of memory-update information in each frame and table view.

Tool-Kit) library, which is open source based on OpenGL [26]. In addition, we utilized the GNU binary utilities to acquire information such as the stack or section, which can be generated by static analysis [27]. Using this information can reduce the area to check for debugging by filtering out unnecessary information, based on the symbol or section information mapped at each address and the memory-update frequency. Using the symbol information, we can distinguish the variables that are not necessary to understand the operation, such as the OS stack for which the memory-update frequency is meaningless or the buffer and timer which are used dynamically [5]. Therefore, the debugging time can be shortened by reducing the fault candidates in accordance with the symbol information. With the processed data mentioned above, it is possible to create two image types using the visualization methods proposed in this paper.

Visualizing $UF_{A,I}$ (Visualizing Total Update Frequency). Figure 8 shows an image containing the total update frequency during the program execution for each address. Each value has a different color according to the number of updates. The higher number of updates has a color close to purple and red, and the lower number of updates has a color close to green and yellow. The areas where no updates occurred are white. The candidates to check for debugging can be reduced by filtering out the variables that have no relevance to the operation or where the update frequency is meaningless.

In Figure 8, ① is related to the stack used in the OS, ② is the buffer area, and ③ is the counter and timer area. Each area has characteristics, such as having dynamic values or being temporarily used as buffers, and the stored data value has meaning while its update information is meaningless. Therefore, these areas can be represented as white to reduce the fault candidates. By using the image in Figure 8, developers can select the focus area based on the total update frequency related to the operation.

Visualizing $U_{A,k}$ in Each Time k (Visualizing Update Information in Each Frame). The memory-update information in each frame is visualized in Figure 9. The left side of the figure is a list of images containing the memory-update information that occurred in every cycle of the system's main

task during the execution. Each image in the list can be expanded to the right side of the image. Gray means that it has no relevance to the operation or no updates that occurred during the execution. Red means an update that occurred in the selected time. White means that an update that occurred during execution, but not in the selected frame. With the list of images, a developer can understand the memory-update phenomenon during execution in the area from Figure 8 that the developer selected to focus. Using this information, the developer can grasp when a fault occurred. The data stored in each frame is also provided in the form of a table.

With the utilization above, developers have the following advantages for understanding the causes of faults generated during the HiL test. First, it is possible to catch both the time point when the fault occurred, using the memory-update information, and also visual information, such as the combination of symbols causing the faults, in environments where source code use is restricted, such as black-box testing. In addition, since the size of the memory data collected periodically is large, it is more effective to intuitively access the cause of the faults using images such as Figures 8 and 9 rather than that provided in the text.

6. Conclusion

In this paper, we proposed a method to collect large amounts of memory data generated inside the ECU by using CAN to monitor the behavior in the HiL test environment, where using a general debugger, utilizing additional system resources, and modifying the hardware are restricted. In the proposed method, we divided the memory into a size that could be sent during the operation cycle, considering the communication environment. Then, we repeated the simulation with the same test script to collect the entire data region, and the collected data was verified. In addition, we provided useful information to access the cause of faults by visualizing data obtained from the memory-update information.

A large amount of data could be transmitted using our method, which does not require hardware modifications or using a debugger and also does not affect the existing operation. Furthermore, in the HiL test environment, where using source code is limited, effective access to the fault was possible by utilizing the images containing the memory-update information. As a result, the debugging time was shortened, compared to finding the faults using text.

For future work, we will dynamically consider the bus occupancy to fully utilize the bus to enhance the transmitting efficiency. With the enhanced efficiency, the number of simulation repetitions can be reduced. Moreover, a more intuitive understanding can be provided when the symbol and section information assigned to each memory address is provided together with the images.

Competing Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the MSIP (NRF-2016R1A2B1014376).

References

[1] P. Braun, M. Broy, F. Houdek et al., "Guiding requirements engineering for software-intensive embedded systems in the automotive industry: the REMsES approach," *Computer Science - Research and Development*, vol. 29, no. 1, pp. 21–43, 2014.

[2] E. Bringmann and A. Krämer, "Model-based testing of automotive systems," in *Proceedings of the 1st International Conference on Software Testing, Verification and Validation (ICST '08)*, pp. 485–493, April 2008.

[3] M. Gomez, "Hardware-in-the-loop simulation," *Embedded Systems Programming*, vol. 14, pp. 38–49, 2008.

[4] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Approach*, Department of Computer Science and Engineering, University of California, 1999.

[5] K. Choi, J. Seo, S. Jang, and J. Lee, "HiL test based fault localization method using memory update frequency," in *Advances in Computer Science and Ubiquitous Computing*, vol. 373 of *Lecture Notes in Electrical Engineering*, pp. 765–772, Springer, Singapore, 2015.

[6] S. Tuohy, M. Glavin, C. Hughes, E. Jones, M. Trivedi, and L. Kilmartin, "Intra-Vehicle Networks: A Review," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 534–545, 2015.

[7] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1222, 2005.

[8] A. Sangiovanni-Vincentelli and M. Di Natale, "Embedded system design for automotive applications," *Computer*, vol. 40, no. 10, pp. 42–51, 2007.

[9] T. Ziermann, S. Wildermann, and J. Teich, "CAN+: a new backward-compatible controller area network (CAN) protocol with up to 16x higher data rates," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, pp. 1088–1093, Nice, France, April 2009.

[10] T. Ziermann and J. Teich, "Electromagnetic Compatibility (EMC) of CAN+," in *GMM-Fachbericht-AmE 2010—Automotive meets Electronics*, VDE, 2010.

[11] G. Cena and A. Valenzano, "FastCAN: a high-performance enhanced CAN-like network," *IEEE Transactions on Industrial Electronics*, vol. 47, no. 4, pp. 951–963, 2000.

[12] Y. Wu, J.-G. Chung, and M. H. Sunwoo, "Design and implementation of CAN data compression algorithm," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '14)*, pp. 582–585, IEEE, Victoria, Canada, June 2014.

[13] Y.-J. Wu and J.-G. Chung, "Efficient controller area network data compression for automobile applications," *Frontiers of Information Technology & Electronic Engineering*, vol. 16, no. 1, pp. 70–78, 2016.

[14] R. Miucic, S. M. Mahmud, and Z. Popovic, "An enhanced data-reduction algorithm for event-triggered networks," *IEEE Transactions on Vehicular Technology*, vol. 58, no. 6, pp. 2663–2678, 2009.

[15] S. Kelkar and Rajkamal, "Control area network based quotient remainder compression-algorithm for automotive applications," in *Proceedings of the 38th Annual Conference on*

IEEE Industrial Electronics Society (IECON '12), pp. 3030–3036, October 2012.

[16] S. Kelkar and R. Kamal, "Boundary of fifteen compression algorithm for controller area network based automotive applications," in *Proceedings of the International Conference on Circuits, Systems, Communication and Information Technology Applications (CSCITA '14)*, pp. 162–167, Maharashtra, India, April 2014.

[17] C. Shin, "A framework for fragmenting/reconstituting data frame in Controller Area Network (CAN)," in *Proceedings of the 16th International Conference on Advanced Communication Technology (ICACT '14)*, pp. 1261–1264, February 2014.

[18] R. Oshana, *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*, Newnes, Oxford, UK, 2013.

[19] S. P. Reiss, "Visual representations of executing programs," *Journal of Visual Languages and Computing*, vol. 18, no. 2, pp. 126–148, 2007.

[20] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pp. 467–477, ACM, Orlando, Fla, USA, May 2002.

[21] H. Shokry and M. Hinchey, "Model-based verification of embedded software," *Computer*, vol. 42, no. 4, pp. 53–59, 2009.

[22] J. S. Keränen and T. D. Räty, "Model-based testing of embedded systems in hardware in the loop environment," *IET Software*, vol. 6, no. 4, pp. 364–376, 2012.

[23] S. Corrigan, "Texas instrument," Application Report, 2008.

[24] J. Lee, K. Choi, S. Y. Jang, and J. Lee, "Data cascading method for the large automotive data acquisition beyond the CAN bandwidth in HiL testing," in *Advances in Computer Science and Ubiquitous Computing*, Lecture Notes in Electrical Engineering, pp. 773–780, Springer, Singapore, 2015.

[25] Trace 32, http://www.lauterbach.com/frames.html?debugger features.html.

[26] VTK-The Visualization Toolkit, http://www.vtk.org/.

[27] GNU Binary Utilities, https://sourceware.org/binutils/docs/binutils/index.html.