Log Data Collection of Real-time Control System using Fault Tree Analysis

Naoya Chujo†, Akihiro Yamashita‡, Nobuyuki Ito‡, Yukihiko Kobayashi‡, and Tadanori Mizuno†

†Faculty of Information Science, Aichi Institute of Technology, Japan
‡Mitsubishi Electronic Engineering Co., Ltd., Japan
ny-chujo@aitech.ac.jp
{Yamashita.Akihiro, Ito.Nobuyuki, Kobayashi.Yukihiko}@ma.mee.co.jp
mizuno@mizulab.net

Abstract - The increasing complexity of embedded systems in information and communication technology causes a problem with locating faults during system failures. One reason for this problem is that complicated systems consist of so many components that basic log data do not contain useful information about abnormal system behavior by faulty components. Since available time resources in real-time systems are limited, we cannot use much time for logging all data to specify the faulty components.

In this paper, we present a logging method of real-time control system using Fault Tree Analysis for locating the faulty components. Fault Tree Analysis is applied for assumed system failures, and then specific data in fault trees are defined to locate the faulty components. Log tasks are scheduled to collect the specified data in cooperation with system tasks. Once the assumed system failure is observed during system operation, the related log tasks wake up and collect the specified data to diagnose system faults. The experimental results have shown that specified data related to faulty components are collected by log task and the overhead for logging is predictable.

Keywords: Fault Tree Analysis, Log Data, Fault Diagnosis, Real-time, Embedded System

1 INTRODUCTION

In recent years, while the embedded software in systems such as those in automobiles or medical devices has grown increasingly complicated, numerous real-time control systems have been developed. These complexities have caused a range of problems, including reduced productivity and increasing difficulty in pinpointing fault origins. Thus, improving the reliability of such systems has become an important objective. Logging data has become a popular method to improve the reliability.

The primary role of the log data associated with faults is to record items such as fault occurrence time and the nature of the fault. In addition, fault-diagnosis functions allow the collection of log data describing the basic status of the system at the fault occurrence time. However, by using only this basic level of log data, it remains difficult to determine the factors that caused the fault to occur.

In this paper, we present a logging method for a real-time control system using a fault tree analysis to locate the faulty components. For assumed system failures, specific data in fault trees are used to locate the faulty components.

To summarize our contribution, we find that log data collection based on fault tree analysis is useful for the identification and tracking of the failed component through our case studies. Further, the time required for data collection is predictable, and the log task is able to be scheduled not to disturb control tasks. Although constructing a fault tree of complex system requires long time, fault trees are assumed to be given in this study.

The remainder of this paper is organized as follows.

In Section 2, we review related work and discuss case studies involving automotive fault diagnosis and reliability improvements by using a fault analysis model. In Section 3, we describe our proposed method to collect log data based on the fault tree analysis for a real-time system. In Section 4, experiments using a miniature car and a motor control system are presented. The results show that the faults of a real-time system can be detected by the proposed method. Moreover, the overhead for collecting log data is evaluated, because predictable overhead is important for a real-time system. In Section 5, we discuss these results. Our conclusions are presented in Section 6.

2 RELATED RESEARCH

Real-time control systems for applications such as automobiles and medical devices require extremely high reliability, and various methods exist for improving system reliability. In this section, we introduce a case study of automotive fault-diagnosis functionality. We also discuss the fault tree analysis (FTA) [1] to improve reliability.

2.1 Fault Diagnosis in Automobiles

The field of automotive fault-diagnosis functionality provides a case study of log data collection in a real-time control system. For example, on-board diagnosis (OBD) [2], which is a tool for diagnosing system status in automobiles, consists of an automatic diagnosis via the computers embedded in automobiles. Most automobiles in service today are equipped with OBD. Figure 1 shows a diagram of OBDII, which is a second-generation OBD system. OBDII monitors and diagnoses Electronic control units (ECUs) via the controller area network (CAN). ECU is an embedded system that controls the automotive electrical and electronic system.

The basic scope of OBD encompasses monitoring, data recording, and communication. Monitoring is checking for the flashing of malfunction indicator lamps (MILs) when rel-

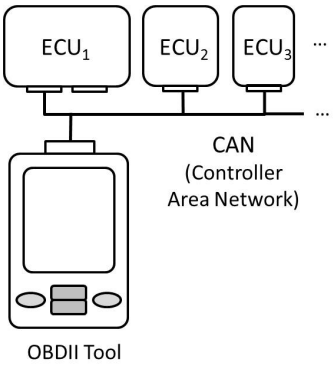


Figure 1: Second-generation on- board diagnosis system (OBDII).

evant items meet fault detection criteria. Data recording and communication are the recording of a code (in the event of a fault) and the specifying of the fault. A diagnostic tool can subsequently be used to read this code.

It is generally believed that monitoring frameworks in OBD systems should grow increasingly sophisticated in the future. Monitoring frameworks include an automotive fault-diagnosis function that allows computers to detect faults.

When an automobile detects a fault, it records diagnostic trouble code (DTC) that encodes information on the sensors involved in the fault, the events that have been diagnosed and the basic status of the automobile. The basic status data is called Freeze Frame Data (FFD).

However, information of the FFD is limited to basic automobile information, such as engine rotation, temperature of cooling water, and O_2 sensor output. Although the FFD includes much necessary information, it is not sufficient for diagnosing the individual controllers used in modern automotive systems. A high-end modern automotive system has dozens of ECUs using an estimated sixty-five million lines of code [3].

Moreover, the real-time nature of controllers makes it difficult to diagnose faults, because the period of a control cycle by sophisticated controllers is on the order of milliseconds. It is much shorter than the period of the FFD, which has a recording cycle time of hundreds of milliseconds, and most typically 500 milliseconds [4], depending on the system.

2.2 Fault Tree Analysis to Improve Reliability

FTA was developed for the reliability assessment and safety analysis of military systems [1]. To date, FTA has been widely applied to various types of industrial plants and transportation systems. In an FTA, the lower event and/or combinations of such events are investigated to determine whether they caused the higher event or the final top-level event, which is undesirable system failure. The tree format, called a fault tree, is defined to express the relation between the lower events and higher events.

As an example of an FTA, we consider the case of car halt. Figure 2 shows the fault tree (FT) diagram corresponding to the halt of an electric vehicle (EV).

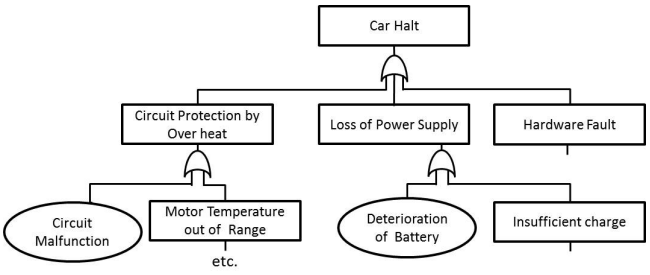


Figure 2: FT diagram for the case of car halt.

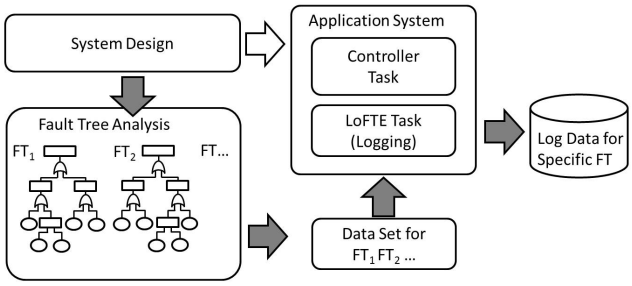


Figure 3: Schematic diagram of the LoFTE method.

We note first that the event at the top of the diagram is the undesirable event within the system. In this case, we positioned car halt as the top-level event. Possible causes for this top-level event include circuit protection by overheat, loss of power supply, and hardware fault. Among these events, possible causes for circuit protection by overheat include circuit malfunction and motor temperature out of range. We proceed in this way to trace the possible causes of each event.

The rectangles in the diagram show intermediate events that could be possible causes for the upper-level events. The circles in the diagram show basic events that could cause system faults.

By specifying events that could cause system faults (top-level events), an FTA enumerates the causes of lower-level events that lead to top-level events (system faults). This enumeration can then be used to analyze the causes and fault events that contributed to the system failure.

FTA was developed as an analysis technique in system design for the prevention of system failure, but it is often used for the purpose of finding the cause after a system failure. In this paper, FTA is used to data collection for the cause analysis of failures in real-time system.

Enhanced FTA approaches to analyze large scale complicated computer-based systems were developed. For example, dynamic fault tree (DFT)[5] is an effective method for the analysis of computer-based systems. DFT provides a means for combining FTA with Markov analysis for sequence dependent problems. DFT method works for fault tolerant computing systems by introducing the functional dependency gates and the spare gates. Another extension of FTA is condition-based fault tree analysis (CBFTA) [6]. CBFTA starts with the known FTA, but by condition monitoring system, CBFTA updates failure rates and applies to the FTA. CBFTA recalculates

periodically the top event failure rate, and then the system reliability is monitored in undergoing system. These enhanced FTA approaches also work for the cause analysis of failures in real-time system.

However, real-time control systems may have the unexpected failure during operation, and some failures are not reproduced in the off-line. We think that it is because of incompleteness or difficulty of modeling control systems including mechanical, electrical, and computer parts. Therefore, there are needs to perform the log data collection and off-line diagnosis.

3 PROPOSED METHOD FOR COLLECTING LOG DATA

In this section, we present our proposed method, which we named *Log data collection using Fault Tree Expansion* (LoFTE) [7]. We then describe our method by using a simple example of a system fault and the FT, after which we demonstrate a case of fault-event identification. Throughout this research, single fault was assumed.

3.1 Philosophy of LoFTE Method

The LoFTE method implements FTA at the system design stage and determines both the collection schedule and the data to be collected at the time of fault detection. Then, during the system operation stage, log data are collected, with proper consideration paid to the real-time nature of the system at the fault detection time. Thus, this method aspires to achieve real-time fault diagnosis by collecting the log data during system operation. More specifically, our method executes the following procedures at the system design stage and during system operation.

Procedure at the system design stage:

1. conduct FTA based on the system design specifications.
2. Within the FT, specify the data to be collected by software.
3. Store collection schedules for the specified data within the control software.

Procedure at the system operation stage:

1. Identify fault(s) that arise during control tasks.
2. Report the log-data-collection task responsible for information associated with the fault(s)
3. The data associated with the fault(s) are then collected as log data based on the relevant scheduling.

We further describe the system by referring to the example depicted in Fig. 3. The results of the FTA at the system design stage are stored as FT_1, FT_2, and so on. In this case, we assume that a fault in FT_2 has occurred and we initiate the log-data collection task. The log-data collection task collects the log data displayed for FT_2.

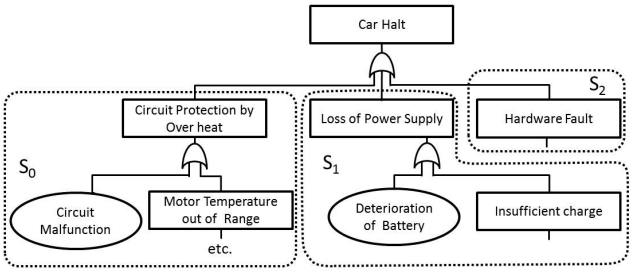


Figure 4: Subtrees for the case of car halt.

3.2 Collection of Log Data

In the LoFTE method, the collection of log data is executed based on information of the fault tree. Once a faulty event is detected, all events in the fault tree should be recorded. However, the log data of a large number of events are not preferable, because the work of collecting log data takes a long time and makes it difficult to analyze faulty events. To this end, the control software is analyzed to determine its module structure [8]. The collection of log data is executed based on the structure [9].

In this subsection, we use the example of the EV discussed in subsection 2.2 as an example of a real-time control system and consider the collection of log data.

The fault tree diagram in Fig. 2 has three subtrees: S_0 of the control circuit module, S_1 of the power supply module and S_2 of the other hardware module. These subtrees are shown in Fig. 4. We assume that the control software for each module is designed to be independent from the others. Therefore, the work of collecting log data for the fault tree is divided into three parts.

4 EXPERIMENTS

In this section, we describe our experiments conducted to test whether it is possible to identify the cause of a fault from the collected sensor data and the FT. The overhead of the proposed method is estimated through experiments.

4.1 Experiment with Miniature Car

In the first experiment, we used a miniature car as an experimental device.

4.1.1 Experimental Device

The experimental device was a 1/10-scale miniature car, Robocar 1/10 (hereafter referred to as RoboCar) for the automotive platform (AP) [10], which was designed to be a research platform for autonomous driving. Figure 5 show a photograph of the experimental device, and Figure 6 show a structural diagram of the RoboCar system.

RoboCar is equipped with a V850/FG4 CPU [11], multiple input devices, including a three-axis acceleration sensor, eight infrared range sensors, a three-axis gyro sensor, two field-effect transitor (FET) temperature sensors, a motor encoder,

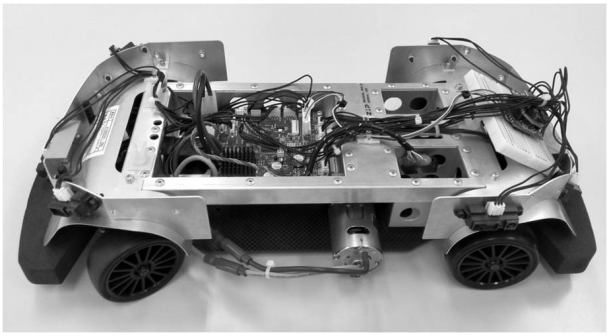


Figure 5: RoboCar 1/10 for AP.

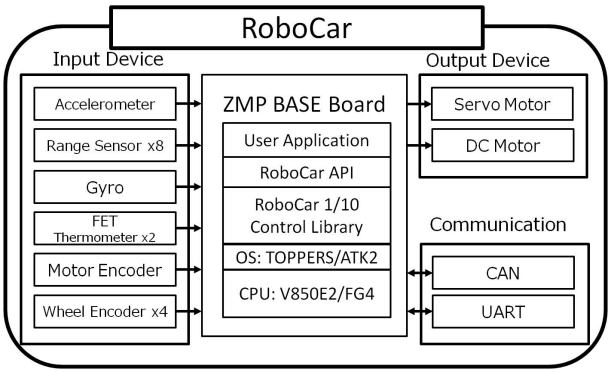


Figure 6: Structure of the RoboCar control system.

and four wheel encoders. Sensor data from all of these input devices are obtained from the RoboCar API. The device is also equipped with two output devices: a servo motor and a DC motor. These devices are controlled by the RoboCar API. The communications specifications correspond to the CAN [12] and the Universal Asynchronous Receiver Transmitter (UART).

4.1.2 Software Used in the Experiment

In this experiment, we used TOPPERS/ATK2 [13], a real-time OS designed for next-generation automotive embedded systems. This OS was designed by the Center for Embedded Computing Systems at Nagoya University (NCES) and was designed to comply with AUTOSAR [14], a standard specification for automotive embedded software.

4.1.3 Experimental System

The experimental system used in this work consists of the RoboCar (the system in which the fault occurs) and a computer that monitors sensor data transmitted from the RoboCar via Bluetooth. Figure 7 shows a schematic diagram of the experimental system[15].

Three tasks were implemented as TOPPERS/ATK2 applications: our proposed LoFTE task, a control task, and a communication task. The LoFTE task collects log data from the sensors installed on the RoboCar. The control task controls the various system actuators on the basis of the sensor data

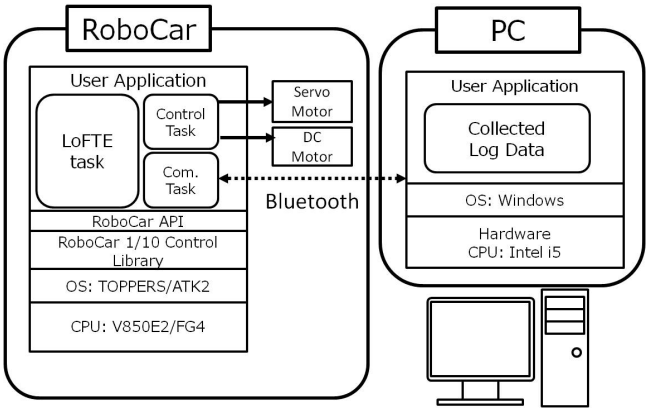


Figure 7: Schematic depiction of the experimental system.

collected by the sensor-data collection task. For example, this task controls the motor torque to ensure that the driving motor maintains a constant velocity. The communication task transmits the data collected to the computer. The system was realized by periodically executing these tasks at a cycle period of 100 msec.

In this experiment, Bluetooth wireless communication was used for the communication. The rate of data transmission via Bluetooth was at 115,200 bits per second (bps) to allow the computer to monitor data transmission from the RoboCar.

4.1.4 Assumed Fault and Experimental Procedure

In the first experiment, the RoboCar traversed a circular track in the clockwise direction until its motion was obstructed manually to bring the RoboCar to a halt. We take the halt of the RoboCar as our fault event in this experiment. We designed one specific fault event for this experiment.

At the system design stage, we anticipate the causes of the RoboCar halt and prepare the fault tree diagram, shown in Fig. 8. The shaded events indicate the causes of the fault assumed in our experiment.

We assigned codes for all events of the fault tree according to the level. The event of Level 0, car halt, was assigned the code, 0x01. Three events of Level 1, circuit protection by overheat, loss of power supply, and hardware fault, were assigned codes, 0x10, 0x11, and 0x12, respectively. Two events of Level 2, circuit malfunction and motor temperature out of range, were assigned codes, 0x20, and 0x21, respectively. Three events of Level 3, cooling fault, overload of motor, and fault of temperature sensor, were assigned codes, 0x30, 0x31, and 0x32, respectively. Two events of Level 4, overcurrent and overvoltage, were assigned codes, 0x40, and 0x41, respectively. The codes of the shaded events of Fig. 8 are listed in Table 1.

In this experiment, we measured the velocity of both wheels powered by the driving motor, the current, and the FET temperature to record the status of the RoboCar. We configured the overheat protection function to go into operation at 80.0 ℃, and the current limit function at 10 A.

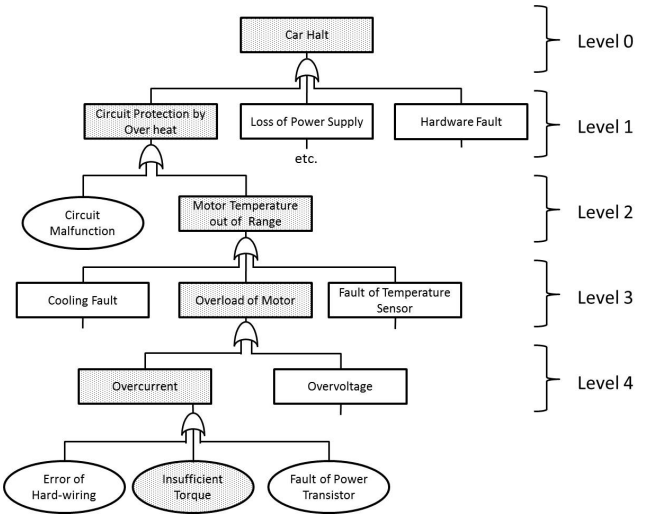


Figure 8: FT diagram for the case of RoboCar halt.

Table 1: Assigned codes for the case of RoboCar halt.

Event	Log Data	Code
Car Halt	Wheel Speed	0x 01
Circuit Protection by Overheat	Flag of Circuit Protection	0x 10
Out of Range of FET Temp.	FET Temperature	0x 21
Overload of Motor	Flag of Overload of Motor	0x 31
Overcurrent	Motor Current	0x 40

4.1.5 Experimental Results

Table 2 shows the experimental results. The RoboCar motion was obstructed at 3'59"7 after measurements began. The driving motor continued to operate after this time, but stopped 10 seconds later.

In the interval prior to 3'59"7, the drive velocity of the RoboCar (which was moving clockwise) was approximately 0.8 m/sec for the left wheel and 0.6 m/sec for the right wheel. At the time the car reached the obstruction, the velocity of both wheels fell to 0.3 m/sec. At 4'00"4, the left wheel velocity jumped to 1.4 m/sec, while the right wheel velocity fell to 0.0 m/sec, which shows the right wheel was locked. At 4'09"7, the driving motor halted and the velocity of both wheels fell to 0.0 m/sec.

The current of the RoboCar ranged from 1.0 A to 4.0 A in the interval prior to 3'59"7, but it jumped to a maximum value of 7.7 A after encountering the obstruction. At 4'00"4, the current rose to 10.2 A, which showed the overcurrent.

The temperature remained roughly constant until 3'59"7, but it jumped to 80.2 ℃ at 4'09"4 after encountering the obstruction. The driving motor halted at 4'09"7, and the temperature remained high.

The event codes 0x40 and 0x31, which corresponded to overcurrent and overload of motor, were detected at 4'00"04. In addition, 0x21and 0x10 , which corresponded to motor

Table 2: Detected events induced by RoboCar halt.

Event	Time	wheel Speed (Left) [m/sec]	wheel Speed (Right) [m/sec]	Current [A]	FET Temp. [℃]	Detected Event Code	Execution Time for Logging [μsec]
Normal	0'00 "0 – 3'59"6	0.8	0.6	1.0~4.0	49.5	-	4
Drive Obstruction	3'59"7	0.3	0.3	7.7	50.5	-	4
Overcurrent and Overload	4'00"4	1.4	0.0	10.2	52.2	0x40, 0x31	124
Out of Range of FET Temp.	4'09"4	1.4	0.0	7.6	80.2	0x40, 0x31 0x21, 0x10	139
Car Halt	4'09"7	0.0	0.0	9.9	79.9	0x40, 0x31 0x21, 0x10 0x01	149

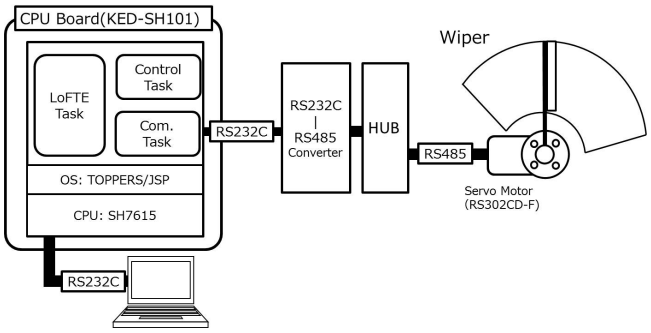


Figure 9: Wiper control system.

temperature out of range and circuit protection by overheat, were detected at 4'09"04. The code 0x01, which corresponded to car halt, was detected at 4'09"07.

In addition, we saw that the execution time for logging increased after detection of the codes. The execution time was 4 μsec at 3'59"7. The execution time jumped to 124 μsec at 4'00"4, and it increased with the number of detected codes.

4.2 Experiment with Wiper Control System

In the second experiment, we used a wiper control system as the experimental device.

4.2.1 Experimental System

Figure 9 shows a structural diagram of the wiper control system. The device is equipped with a CPU board (KED-SH101), a servo motor, and a monitoring computer. These are connected via RC232C and RS485 networks. The servo motor can be controlled from KED-SH101 via the networks.

In this experiment, we used TOPPERS/JSP [16], a real-time kernel designed for embedded systems.

Three tasks were implemented as TOPPERS/JSP applications (the LoFTE task, a control task, and a communication task), and the system was realized by periodically executing these tasks. The LoFTE task was executed every 700 ms. The LoFTE task collects log data from the servo motor, and the data are sent to the monitoring computer.

4.2.2 Assumed Fault and Experimental Procedure

For this experiment, we designed two specific fault events that cause a wiper halt. The moving wiper operation was obstructed manually, and it caused the servomotor to increase the motor current to keep it moving. Because the motor temperature increased due to the increased current, it finally brought about a wiper halt. We take the obstruction as the first fault event in this experiment.

As our second fault event, we designed a communication fault event. When the wiper system was moving to the left and right, the communication was interrupted intentionally by setting the disable flag of the communication register, bringing no response from the servo motor. The interruption also caused a wiper halt.

At the system-design stage, we anticipate the causes of a wiper halt and prepare the fault tree diagram shown in Fig. 10. The shaded events indicate the causes of the fault assumed in our experiment.

We assigned the codes for all events of the fault tree according to level. All of the assigned codes are listed in Table 3. The event of Level 0, wiper halt, was assigned code 0x01. Three events of Level 1, circuit protection by overheat, communication fault, and hardware fault, were assigned codes 0x10, 0x11, and 0x12, respectively. Six events of Level 2, motor temperature out of range, circuit malfunction, check sum error, parameter read error, response time out, and negative acknowledge, were assigned codes 0x20, 0x21, 0x22, 0x23, 0x24, and 0x25, respectively. Six events of Level 3, cooling fault, overload of motor, fault of temperature sensor, no response from servo motor, excess number of retries, and disconnection of network, were assigned codes 0x30, 0x31, 0x32, 0x33, 0x34, and 0x35, respectively. Four events of Level 4, overcurrent, overvoltage, loss of motor power supply, and communication disable flag, were assigned codes 0x40, 0x41, 0x42, and 0x43, respectively.

In this experiment we measured current, voltage, temperature and the control register of the servo motor. We configured the overheat protection function to go into operation at 60 ℃, and we set the maximum current at 100 mA. The communication control register and the number of retries were monitored in this experiment.

Table 3: Assigned codes for the case of wiper halt.

Event	Code
Wiper Halt	0x01
Circuit Protection by Over heat	0x10
Communication Fault	0x11
Hardware Fault	0x12
Motor Temperature out of Range	0x20
Circuit Malfunction	0x21
Check Sum Error	0x22
Parameter Read Error	0x23
Response Time Out	0x24
Negative Acknowledge	0x25
Cooling Fault	0x30
Overload of Motor	0x31
Fault of Temperature Sensor	0x32
No Response from Servo Motor	0x33
Excess Number of Retries	0x34
Disconnection of Network	0x35
Overcurrent	0x40
Overvoltage	0x41
Loss of Motor Power Supply	0x42
Communication Disable Flag	0x43

Table 4: Detected events induced by wiper obstruction.

	Time	Current [mA]	Temp. [℃]	Detected Event Code	Execution Time for Logging [msec]
No Operation	0'00 "00 – 0'3"32	0	41	-	1
Normal Operation	0'03"99 – 0'07"08	12~14	42	-	1
Overcurrent	0'07"68	611	42	0x40, 0x31	1
Wiper Halt	1'14"45	568	60	0x40, 0x31 0x20 0x10, 0x01	2

4.2.3 Experimental Results

Table 4 shows the experimental results for the first fault event. The wiper motion was started at 0'03"99 after measurements began. Then, the wiper's motion was manually obstructed at 0'07". The motor continued to operate after this time, but stopped at 1'14"45.

The current of the wiper system was in the range from 12 mA to 14 mA in the interval from 0'03"99 to 0'07"08, but it jumped to a value of 611 mA after encountering the obstruction, which showed the overcurrent because the current limit was 100 mA. The overcurrent was kept to 1'14"45.

The temperature remained roughly constant at 41 ℃ or 42 ℃ until 0'07"08, but it rose to 60 ℃ at 1'14"45 after encountering the obstruction.

Event codes 0x40 and 0x31, which corresponded to overcurrent and overload of motor, were detected at 0'07"68. In addition, 0x20, 0x10, and 0x01, which corresponded to temperature exceeding limit (60 ℃), circuit protection by overheat, and wiper halt, were detected at 1'14"45.

The execution time for logging was 1 msec after detecting two event codes at 0'07"68, then it increased to 2 msec when detecting five event codes at 1'14"45. Because the time resolution of TOPPERS/JSP kernel is 1 msec, we could not see that the execution time increased with the number of detected codes.

Table 5 shows the experimental results for the second fault event. The wiper started to swing at 0'07"18 after measurements began. The communication was obstructed at 0'16"47 after measurements began. The wiper halted at 0'17"07.

In the interval prior to 0'15"79, the register code of the communication register was 0x30 (00110000), which showed

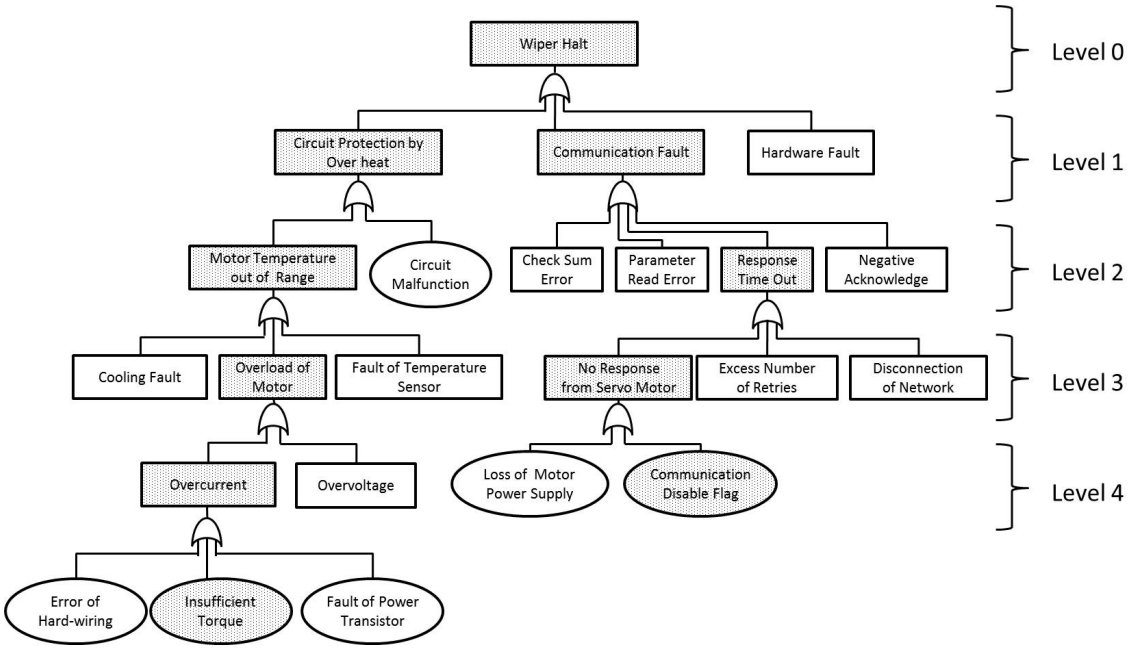


Figure 10: Assumed fault in wiper control system.

Table 5: Detected events induced by communication error.

	Time	Register Value	Num. of Retry	Detected Event Code	Execution Time for Logging [msec]
No Operation	0'00 "00 – 0'7"18	0x30	0	-	1
Normal Operation	0'07"19 – 0'15"79	0x30	0	-	1
Communication Fault	0'16"47	0x00	0	0x43, 0x33	1
Wiper Halt	0'17"07	0x00	6	0x43, 0x33 0x24 0x11, 0x01	2

that the transmit enable bit and the receive enable bit were set, and the communication was enabled. At 0'16"47, the register code changed to 0x00 (00000000), which showed that the communication was disabled. At 0'17"07, the number of retries for the communication reached 6, which showed that the maximum number of retries was exceeded.

Event codes 0x43 and 0x33, which corresponded to disable flag of communication and no response from servo motor, were detected at 0'16"47. In addition, 0x24, 0x11, and 0x01, which corresponded to response timed out, communication fault, and wiper halt, were detected at 0'17"07.

5 DISCUSSIONS

Our experiments confirmed that the LoFTE method was capable of logging the sequences of faulty events up to the top-level event. The sequences of faulty events provide useful information for identifying the cause of the fault.

In the first experiment using the RoboCar, we see first that over current and overload of the motor can be detected, then the events on the path of the fault tree can be detected, and finally the RoboCar halt (the top-level event) can be detected.

In the second experiment using the wiper control system, we see that the communication disable flag (the basic event) and no response from the servomotor can be detected as the second fault event. After that, the events on the path of the fault tree can be detected, and finally the wiper halt (the top-level event) can be detected.

Regarding the load of LoFTE tasks, the maximum execution time was estimated at 149 μsec. This time was less than 1% of that of the control cycle period, 100 msec in the first experiment. However, more importantly, the overhead of real-time system was predictable [17]. Because the execution time for logging increased with the number of the detected codes, the load of the LOFTE task was predictable by the heights of the fault trees.

We assumed a single fault at a time in this study, and so the order of logged events was straightforward on the path of the fault tree. However, in the case of multiple faults, the order of logged events would be complicated.

It should be noted that constructing the complete FT of a complex system is difficult. It is also difficult to evaluate the validity of FT. However, even the FT is not complete, it does not mean useless. The incomplete FTs would have unexpected events or wrong edges between events. If the unexpected event had influence on system failure, they would have implicit edges to some FTs. Such events are considered to be detectable as the other events of FT. Otherwise, the events have no influence on the system, and they are negligible. In the case of the wrong edges, they make the fault analysis difficult, and another logging is necessary to verify the edges between the events.

For this reason, our future work will be to access data sets of the system for logging through remote networks, as shown in Fig. 11. We would be able to access real-time controllers by wireless communication and modify data sets for logging

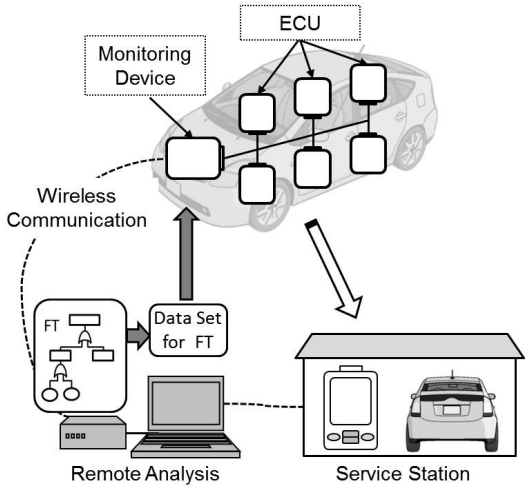


Figure 11: Remote log collection system.

unexpected faults depending on the designer's request. Fault diagnosis of real-time controllers through remote networks would be available. After detecting the faulty device or software, the system could be fixed at service stations or by a wireless network. This capability would reduce the problem of designing complete fault trees of complex systems prior to market release.

6 CONCLUSIONS

In this study, we proposed the LoFTE (Log data collection using Fault Tree Expansion) method for log data collection using a fault tree analysis. We then conducted experiments involving a miniature car and a wiper control system.

The results of our experiments indicate that the LoFTE method is capable of logging the sequences of faulty events. Therefore, it is useful for identifying the cause of a fault. The load of the logging task is small and predictable by using fault trees. The log task is able to be scheduled without disturbing control tasks.

Our future work will be to access and control data sets for the logging of fault trees through remote network.

ACKNOWLEDGMENTS

This research is supported by JSPS KAKENHI Grant Number 26330074.

We extend our deepest gratitude to Associate Professor Shinya Honda and the members of the Center for Embedded Computing Systems at Nagoya University (NCES), who provided extensive support and resources regarding TOPPERS/ATK2 during the completion of this work.

We are also particularly grateful for the assistance and support given by Shogo Fukuoka, Hiroki Kitagawa and Kazuhiro Tsujita.

REFERENCES

[1] N. Leveson, Safeware, System Safety and Computers, ACM, pp. 305-313 (1995).

[2] J. Shaeuffele, and T. Zurawka, Automotive Software Engineering - Principles, Processes, Methods and Tools, SAE International, pp. 118-125 (2005).

[3] Mentor Graphics Inc., "News and Views 2014 Spring/ Vol. 9," accessed June 13 (2015). http://www.mentorg.co.jp/news_and_views/automotive/2014/spring.html

[4] Toyota Motor Corp., Toyota Prius New Model Reference and Repair Manual, Parts Number NM12B1J, pp. IN-36-38 (2010).

[5] R. Gulati and J. Dugan, "A Modular Approach for Analyzing Static and Dynamic Fault Trees," IEEE Proc. of Reliability and Maintainability Symposium, pp. 57-63 (1997).

[6] D. Shalev and J. Tiran, "Condition-based Fault Tree Analysis (CBFTA): A New Method for Improved Fault Free Analysis (FTA), Reliability and Safety Calculations," Reliability Engineering and System Safety, Vol. 92, No. 9, pp. 1231-1241 (2007).

[7] N. Chujo, A, Yamashita, N. Ito, Y. Kobayashi, and T Mizuno, "Real-time Log Collection Scheme using Fault Tree Analysis," Proceedings of International Workshop on Informatics 2015, pp. 109-114 (2015).

[8] M. Takahashi, "A Study of Fault Tree Analysis for Embedded Software," IPSJ Technical Reports, Vol. 2013-SE-182 No. 24, pp. 1-8 (2013).

[9] S. Fukuoka, et al., "Scheduling for Logging of Real-time Control Systems," Proceedings of the 76th National Convention of IPSJ, No. 1, pp. 59-60 (2014).

[10] ZMP Inc., "RoboCar 1/10 for AP (Automotive Platform)," accessed June 13 (2015). https://www.zmp.co.jp/products/robocar-110_package_option#ap

[11] Renesas Electronics, "Data Sheet V850E2/FG4 32-bit Single-Chip Microcontroller (2013)," accessed June 13 (2015). http://documentation.renesas.com/doc/DocumentServer/R01DS0139ED0100_FG4.pdf

[12] Robert Bosch GmbH, "CAN Specification Version 2.0 (1991)," accessed June 13 (2015). http://www.kvaser.com/software/7330130980914/V1/can2spec.pdf

[13] TOPPERS Project, "TOPPERS/ATK2," accessed June 13 (2015). https://www.toppers.jp/en/atk2.html

[14] S. Frst, "AUTOSARTechnical Overview," AUTOSAR Open Conference. (2011).

[15] H. Kitagawa, et al., "Logging for Fault Diagnosis of Realtime Control System," Proceedings of Workshop on Informatics 2014, pp. 158-164 (2014).

[16] TOPPERS Project, "TOPPERS/JSP kernel," accessed June 13 (2015). http://www.toppers.jp/en/jsp-kernel.html

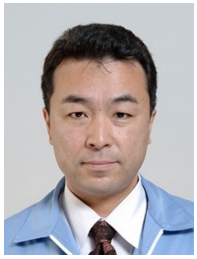
[17] P. Dodd and C. Ravishankar, "Monitoring and Debugging Distributed Real-time Programs," Software, Practice and Experience 22.10 pp. 863-877 (1992).

(Received September 30, 2015)
(Revised March 7, 2016)

Naoya Chujo received his M.S. degree in information science in 1982 and Ph.D. degree in electrical engineering in 2004 from Nagoya University, Japan. He is a professor in the Faculty of Information Science of Aichi Institute of Technology, Japan. His research interests are in the area of embedded system and automotive electronics. He is a member of IEEE, the Information Processing Society of Japan (IPSJ), the Institute of Electronics, Information and Communication Engineers (IEICE), the Institute of Electrical Engineers of Japan (IEEJ), and Informatics Society.

Akihiro Yamashita received the B.E and M.E. degree in Precision Engineering from the Kyoto University, Japan in 1977 and 1979, respectively. In 1979, he joined Mitsubishi Electric Corp. He received the M.E. degree in Mechanical Engineering from the Carnegie Mellon University, USA, in 1987. In 2012, he joined Mitsubishi Electric Engineering Corp. Since 2014, he has been a graduate school student at Shizuoka University, Japan. His research interests include control engineering and computer science. He is a member of IPSJ.

Nobuyuki Ito joined Mitsubishi Electric engineering Co.,Ltd. in 1979. He's a manager of the drive system control engineering department. His expertise field is in the factory automation for which a servo control system was utilized.



Yukihiko Kobayashi joined Mitsubishi Electric engineering Co., Ltd. in 1986. He is a senior engineer of the servo software engineering section. His specialized field is built-in software.

Tadanori Mizuno received the B.E. degree in Industrial Engineering from the Nagoya Institute of Technology in 1968 and received the Ph.D. degree in Engineering from Kyushu University, Japan, in 1987. In 1968, he joined Mitsubishi Electric Corp. From 1993 to 2011, he had been a Professor at Shizuoka University, Japan. From 2011 to 2016, he had been a Professor at the Aichi Institute of Technology, Japan. Since 2016, he is an Affiliate Professor at the Aichi Institute of Technology, Japan. His research interests include mobile computing, distributed computing, computer networks, broadcast communication and computing, and protocol engineering. He is a member of Information Processing Society of Japan, the Institute of Electronics, Information and Communication Engineers, the IEEE Computer Society and Consumer Electronics, and Informatics Society.