Fault-Based Interface Testing Between Real-Time Operating System and Application

Ahyoung Sung, Jina Jang and Byoungju Choi
Dept. of Computer Science and Engg., Ewha University, Seoul, 120-750, Korea
{aysung, jajang}@ewhain.net, bjchoi@ewha.ac.kr

Abstract

Testing interfaces of an embedded system is important since the heterogeneous layers such as hardware, OS and application are tightly coupled. We propose the mutation operators in three respects, 'When?', 'Where?' and 'How?' in order to inject a fault into RTOS program when testing interface between RTOS and application. Injecting a fault without affecting the RTOS in run-time environment is the core of proposed mutation operators. We apply the mutation operators to interface testing during the integration of RTOS and application in the industrial programmable logic controller.

1. Introduction

RTOS (Real Time Operating System) manages an embedded system by executing applications. To test embedded software such as RTOS and application is necessary before the permanent mount of embedded software [1].

RTOS consists of kernel and system tasks. The system tasks are executable control units on kernel that are responsible for running the RTOS. They are tightly coupled with kernel, hardware devices and applications. RTOS manages an application as a system task. Therefore, it is imperative to test interface between RTOS and application. RTOS and application communicate via RS-232C protocol defined as a standard by EIA (Electronic Industries Alliance) [2]. It is also known as UART (Universal Asynchronous Receiver/Transmitter).

In this paper, we test interfaces between RTOS and application by using system tasks on kernel. Here, an interface refers to a gateway that controls the communication between two different layers in an embedded system [3]. This interface is the focal point for monitoring and debugging an embedded system where the heterogeneous layers are tightly coupled. Also, this interface becomes the criteria for test coverage that is used to select test cases [3]. We define it as interface testing.

Generally, mutation analysis is a fault-based testing technique that helps the tester create a set of test cases to detect specific, predetermined type of faults [4]. When we test software by focusing on a small restricted class of faults, we can expect to detect more complicated faults as well, giving us confidence that fault-based testing strategies can provide effective ways to test software [5]. Experimental data shows that faulty versions of a program, called mutants generated by applying mutation operators are similar to real faults [20].

Mutation operators generate mutant programs by injecting a fault into the FIT (Fault Injection Target) of a source program. In other words, the mutant programs are generated by changing the FIT syntactically. Other studies have already developed mutation operators for procedural programs, object-oriented programs and component-based programs by deciding the FIT carefully [6, 7, 8, 9].

It is important to determine when to inject a fault since RTOS is time-dependent [1]. It is also important to determine the FIT of RTOS because locations for injecting a fault are limited. If you modify the locations for controlling the entire embedded system such as kernel, it would cause the system to stop. Therefore, the fault injection time and the fault injection target for RTOS should be carefully considered when developing the mutation operators. In this paper, we propose the mutation operators in three respects, 'When?', 'Where?' and 'How?' in order to inject a fault into RTOS program when testing interface between RTOS and application.

This paper consists of the following sections: Section 2 describes the interfaces between RTOS and application. Section 3 accounts for the mutation operators. Section 4 describes the empirical study.

Finally, section 5 discusses the conclusion and future work.

2. Interface between RTOS and application

As shown in Figure 1, an embedded system consists of hardware layer, OS layer and application layer [10]. In case of RTOS, OS layer is subdivided into system task layer and kernel layer. Interface between RTOS and application is defined according to the RS-232C protocol.

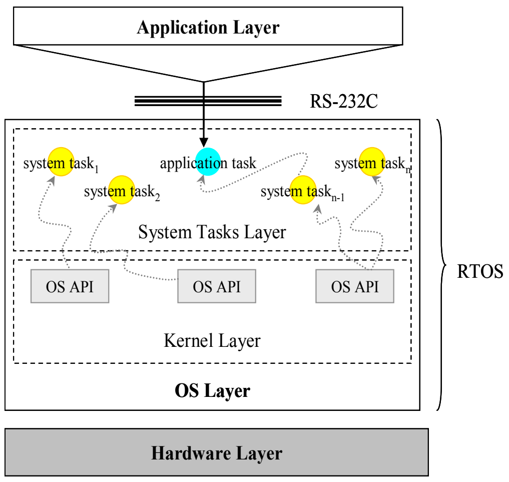


Figure 1. Architecture of an Embedded System

Application communicates with RTOS by sending headers and data, based on the defined RS-232C communication protocol. Figure 2 represents the process of RS-232C communication between RTOS and application in a sequential manner. The parentheses in Figure 2 represent the RS-232C protocol in each communication step as following:

Step① RTOS enables a queue by calling OS API (Application Program Interface) 'OSQPend()'. The queue is used to receive-communication. Then, the RTOS transmits ACK or NAK using semaphore by calling OS API 'OSSemPend()'.

Step② After the application sends the starting header, RTOS transmits ACK or NAK.

Step③ After the application sends the headers, RTOS transmits ACK or NAK.

Step④ If RTOS receives data or text from application, checksum is calculated using the received header information. Then, RTOS transmits ACK or NAK.

Step⑤ When the application has no more data to send, it sends EOT (End of Text).

As shown in Figure 2, RTOS complies with the standard RS-232C protocol. RTOS has interfaces with *rx_symbol_name* during the receiving phase while RTOS has interfaces with *tx_symbol_name* during the transmitting phase. Here, *rx_symbol_name* and *tx_symbol_name* including the prefix '*rx*' and '*tx*' are global variables in the RTOS source code.

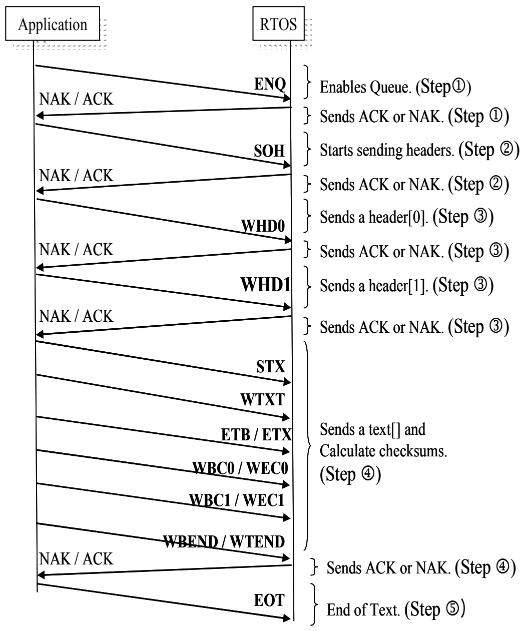

Application
RTOS
ENQ
Enables Queue. (Step①)
NAK / ACK
Sends ACK or NAK. (Step ①)
SOH
Starts sending headers. (Step ②)
NAK / ACK
Sends ACK or NAK. (Step ②)
WHD0
Sends a header[0]. (Step ③)
NAK / ACK
Sends ACK or NAK. (Step ③)
WHD1
Sends a header[1]. (Step ③)
NAK / ACK
Sends ACK or NAK. (Step ③)
STX
WTXT
ETB / ETX
WBC0 / WEC0
WBC1 / WEC1
WBEND / WTEND
Sends a text[] and Calculate checksums. (Step ④)
NAK / ACK
Sends ACK or NAK. (Step ④)
EOT
End of Text. (Step ⑤)

Figure 2. RS-232C Communication Protocol

The application received from RS-232C communication runs with other system tasks in a form of 'application task' which is managed by 'Task Management' API [11, 12] provided by the kernel. Hence, 'application task' has 'Download', 'Run', 'Stop' and 'Delete' states.

Table 1 shows both RTOS execution path based on RS-232C and interfaces to be monitored for each state of 'application task'. As shown in Table 1, the execution path consists of the executed RS-232C protocol and the corresponding OS API.

Table 1. Interface between RTOS and application

State of app. task	Execution path on RTOS	Interface
Download	ENQ-SOH-WHD0-WHD1-STX-WTXT-ETB-WBC0-WBC1-WBEND-data_load()-EOT	*rx_symbol_name* *tx_symbol_name*
Run	ENQ-SOH-WHD0-WHD1-STX-WTXT-ETX-WEC0-WEC1-WTEND-task_fork()-EOT	*rx_symbol_name* *tx_symbol_name*

State of app. task	Execution path on RTOS	Interface
Stop	ENQ-SOH-WHD0-WHD1-STX-WTXT-ETX-WEC0-WEC1-WTEND-task_kill()-EOT	*rx_symbol_name* *tx_symbol_name*
Delete	ENQ-SOH-WHD0-WHD1-STX-WTXT-ETX-WEC0-WEC1-WTEND-task_delete()-EOT	*rx_symbol_name* *tx_symbol_name*

3. Mutation operators

3.1. Problems in injecting a fault into RTOS

Mutation operators generate mutation programs by changing FIT syntactically. Determining the appropriate time and the location for fault injection is critical for two reasons: RTOS is time-dependent software, and FIT of RTOS is limited.

Figure 3 represents the failed situation for communication between RTOS and application in the emulator based run-time environment. Negligence of time and target for fault injection causes 'checksum error' as shown in Figure 3. Here, checksum refers to calculated value for communication between RTOS and application in run-time environment where target board is mounted. 'Checksum error' means abnormal termination of the running RTOS and application on the target board when the calculation result is different.

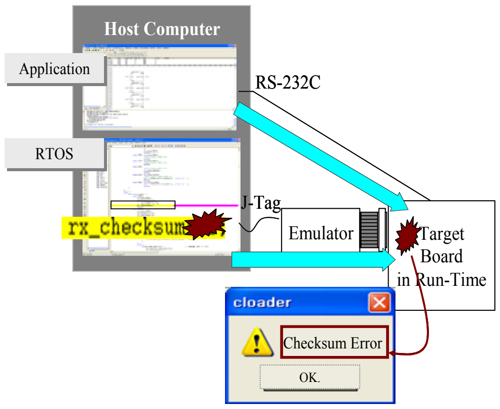


Figure 3. Checksum error of RTOS

3.2. Mutation operators

In order to prevent unexpected errors such as checksum error, we cannot apply the existing methods for generating mutant programs by syntactic change of the FIT. We should inject a fault without affecting the RTOS in run-time environment. We call it 'mutant generation by changing FIT semantically'. In this paper, we propose the mutation operators in three respects, 'When to inject a fault into RTOS?', 'Where to inject a fault into RTOS?' and 'How to inject a fault into RTOS?'.

- ***When***: It is not possible to inject a fault and to monitor the behavior at any time since RTOS is time-dependent software. The analysis of 'When?' is required in order to determine the time for fault injection.
- ***Where***: RTOS source code that developers can modify is limited. If you change the locations to control the entire embedded system such as kernel, it makes the entire system stop. Therefore, it is imperative to identify where in RTOS source code can be modified.
- ***How***: A method to generate faulty versions of program is necessary, especially when regarding how to change the program code.

3.2.1. When: Fault injection time. Injecting a fault into RTOS during the RS-232C communication causes 'checksum error' in most of times. It is important to identify possible time for injecting a fault into RTOS during the RS-232C communication.

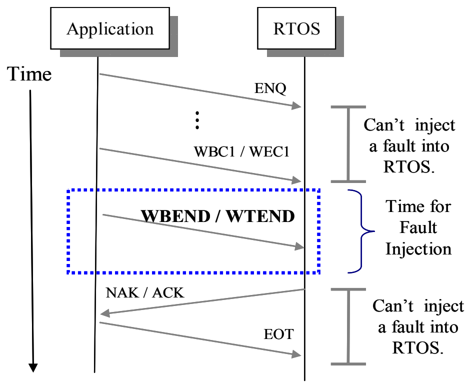


Figure 4. Fault injection time

Figure 4 is a partial capture of RS-232C communication protocol that is fully described in Figure 2. As shown in Figure 4, fault injection to RTOS is possible only 'while WBEND or WTEND is in execution' or 'prior to the transmission of ACK or NAK of corresponding protocol'. This moment is also when checksum calculation for the communication between two layers is complete and about to transmit 'EOT' after completing the RS-232C communication.

If fault injection occurs at other times, RTOS generates 'checksum error' and comes to a halt.

3.2.2. Where: Fault injection target. Fault Injection Target (FIT) should be the interface of RTOS that is affected by RS-232C communication protocol. The interface gets formed through *rx_symol_name* in the event of receive-communication while it gets formed through *tx_symbol_name* in the event of transmit-communication.

Table 2 describes the *rx_symbol_name* and *tx_symbol_name* that correspond to RS-232C communication protocol in more details. Steps in Table 2 are the process of RS-232C communication mentioned in Section 2. Here, Step④ is subdivided into Step④a: 'STX ~ WBC1 or STX ~ WBC0' and Step④b: 'WBEND ~ NAK or ACK' or 'WTEND ~ NAK or ACK'.

Table 2. Fault injection target

Interface	RS-232C Protocol					
	Step ①	Step ②	Step ③	Step ④a	Step ④b	Step ⑤
rx_mode	○	○	○	○	◉	○
rx_header[]	-	-	○	○	◉	○
rx_bcs	-	○	○	○	○	○
rx_checksum	-	○	○	○	○	○
rx_text[]	-	-	-	○	○	○
rx_text_ptr	-	-	-	○	○	○
tx_return	○	○	○	○	○	○

-: Not Available ◉: Fault Injection Target
○: Corresponding Interface to the RS-232C Protocol

As shown in Table 2, *rx_mode* and *rx_header[]* are the FITs among the identified interfaces because they are the modifiable locations in RTOS program during the time of fault injection. *rx_mode* saves next protocol to be sent and *rx_header[]* saves the received header information. These two are modifiable whereas *rx_bcs*, *rx_checksum*, *tx_return*, *rx_text[]* and *rx_text_ptr* are not during the time of fault injection.

In other words, if you inject a fault into *rx_symbol_name* or *tx_symbol_name* other than *rx_mode* and *rx_header[]*, RTOS generates 'checksum error'. It is because *rx_bcs*, *rx_checksum* and *tx_return* are used in the calculation of the checksum for the RS-232C communication between two layers and *rx_text[]* and *rx_text_ptr* are application data themselves.

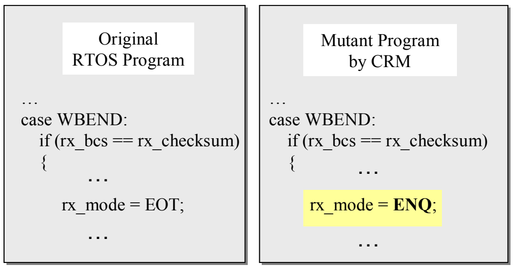
3.2.3. How: Fault injection method. Faulty versions of the program, called mutant programs, are generated by semantic change of the FIT of RTOS program at a proper time.

Figure 5 shows how to generate a faulty program with two mutation operators, CRM and CRH. CRM and CRH operators change the value of *rx_mode* and *rx_header[]* respectively upon the completion of checksum calculation for the RS-232C communication, and prior to the transmission of 'EOT'.

```
Generate_Mutant_Programs (Time, Interface)
{
 Time = Execution Time according to the RS-232C
         communication;
 FIT = Interface;

 while ( Time for sending 'WBEND or WTEND'
          < Time < Time for sending 'ACK or NAK' of
          'WBEND or WTEND')
 {
    switch(FIT)
    {
      /* CRM: Change the value of Rx_Mode */
        case 'rx_mode':
              change the value of rx_mode;
              break;

      /* CRH: Change the value of Rx_Header[] */
        case 'rx_header[]':
              change the value of header;
              break;

    }  /* End of switch */
 }  /* End of while */
}  /* End of Generate_Mutant Programs() */
```

Figure 5. Mutation operators

CRM generates a faulty program by changing the value of *rx_mode*. The execution path in RTOS program is changed according to the value of rx_mode since *rx_mode* determines the execution path of RS-232C communication.



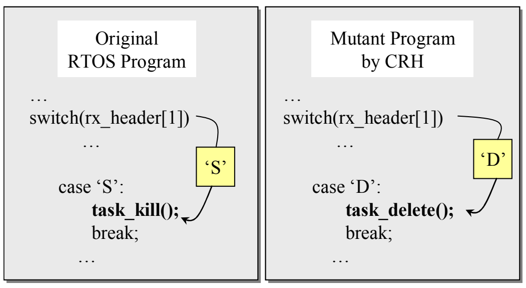
```
…
case WBEND:
   if (rx_bcs == rx_checksum)
   {
        …
      rx_mode = EOT;
        …
```

```
…
case WBEND:
   if (rx_bcs == rx_checksum)
   {
        …
      rx_mode = ENQ;
        …
```

Figure 6. An example of CRM

Taking an example of CRM, Figure 6 is a part of the RTOS source code that implements RS-232C communication. The faulty program Figure 6 is generated by changing from 'EOT' to 'ENQ' while 'WBEND' in execution.

CRH generates a faulty program by changing the ASCII header value in *rx_header[]*. Received *rx_header[]* determines RTOS service and hence, the change in the value of *rx_header[]* changes the OS API to call.



```
…
switch(rx_header[1])
    …
    case 'S':
        task_kill();
        break;
    …
```

```
…
switch(rx_header[1])
    …
    case 'D':
        task_delete();
        break;
    …
```

Figure 7. An example of CRH

Taking an example of CRH, Figure 7 is a part of the RTOS source code that implements RS-232C communication. Since the value of *rx_header[1]* has changed from 'S' to 'D', task_delete() is called at the times when task_kill() is supposed to be called.

4. An empirical study

We have applied the proposed mutation operators to the RTOS in the industrial PLC (Programmable Logic Controller) [13] based on the TI TMS320C32-60 DSP (Digital Signal Processor) board [14]. PLC is the embedded system that is broadly used in the industry such as nuclear power plant, railroad control system and production line where the systems require real-time processing. PLC application is written in programming language such as FBD (Function Block Diagram) defined in the IEC (International Electro technical Commission) standard 61131-3 [15].

The RTOS consists of the micro/C-OS kernel [12] and five system tasks including *Startup task*, *Shell task*, *Diagnosis task*, *LoaderRxrdy task* and *Loader_Service task* [16]. The system tasks download, execute and control the application programs using the RS-232C communication [16].

As shown in Table 3, the number of total lines, file size and programming language for the target RTOS and seven applications are listed. The generated applications had various sizes from 1KB to 971KB due to the limited space of RAM. The maximum size of application could not exceed 1MB on the TMS320C32-60 DSP board [14].

Table 3. RTOS and applications

Target SW	# of Total Lines	File Size	Programming Language
RTOS	9446	2.08 MB	In-line assembly, C language
App.1	4	1 KB	FBD
App.2	301	40 KB	FBD
App.3	1501	197 KB	FBD
App.4	3001	394 KB	FBD
App.5	4501	592 KB	FBD
App.6	6001	700 KB	FBD
App.7	7501	971 KB	FBD

In this empirical study, we generated faulty RTOS programs by applying CRM and CRH to the RTOS. We performed fault-based interface testing by applying CRM and CRH during integration of RTOS and application in the PLC. Here, the fault-based interface means that the identified interface has a fault generated by CRM or CRH.

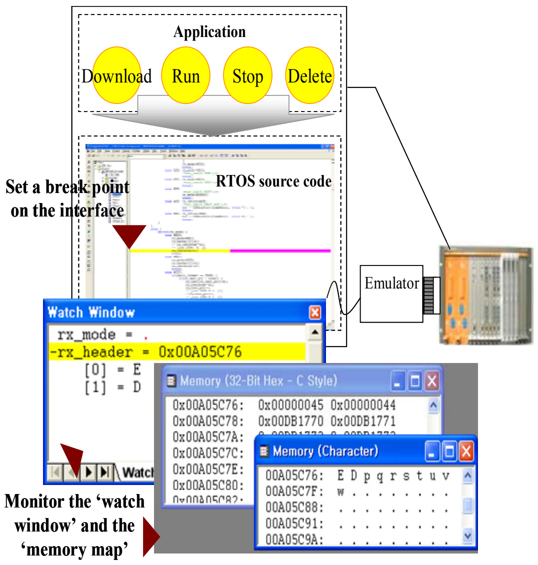
We performed an interface testing based on run-time monitoring [17, 18]. As shown in Figure 8, we stopped running RTOS by setting break points on the interfaces and monitored the current results. We determined 'pass' if the monitored results satisfied the expected output, and we did 'fail' otherwise. To monitor the results, we used the 'watch window' and the 'memory map' of Code Composer [19] that is supported by the TI DSP board.

Table 4. The number of interfaces between RTOS and application

rx_ mode	*rx_ header []*	*rx_ bcs*	*rx_ chec ksum*	*rx_ text[]*	*rx_ text_ ptr*	*tx_ return*	Total
11 (1)	17 (1)	10 (-)	10 (-)	6 (-)	6 (-)	5 (-)	65 (2)

(): # of Fault Injection Target - : Not Available

As shown in Table 4, there were 65 interfaces for the interface between the target RTOS and application, such as *rx_mode*, *rx_header[]*, *rx_bcs*, *rx_checksum*, *rx_text[]*, *rx_text_ptr* and *tx_return*.


Application
Download
Run
Stop
Delete
Set a break point on the interface
RTOS source code
Emulator
Watch Window
Memory (32-Bit Hex - C Style)
Memory (Character)
Monitor the 'watch window' and the 'memory map'

```
rx_mode = .
-rx_header = 0x00A05C76
    [0] = E
    [1] = D
```

```
0x00A05C76:  0x00000045 0x00000044
0x00A05C78:  0x00DB1770 0x00DB1771
0x00A05C7A:
0x00A05C7C:
0x00A05C7E:
0x00A05C80:
```

```
00A05C76:  E D p q r s t u v
00A05C7F:  w . . . . . . . .
00A05C88:  . . . . . . . . .
00A05C91:  . . . . . . . . .
00A05C9A:  . . . . . . . . .
```

Figure 8. Test Environment

The result of interface testing during the integration of RTOS and application is as follows:

First of all, we performed an interface testing for the original RTOS program and seven applications. For each state of application tasks including, 'Download', 'Run', 'Stop' and 'Delete', we performed the testing to cover 65 interfaces for 100%.

In case of 'Download', the tested path was in order of 'ENQ – SOH – WHD0 – WHD1 – STX – WTXT – ETB – WBC0 – WBC1 – WBEND – data_load() – EOT'. In case of 'Run', 'Stop' and 'Delete', the order was 'ENQ – SOH – WHD0 – WHD1 – STX – WTXT – ETX – WEC0 – WEC1 – WTEND – OS API – EOT'. Here, OS API was 'task_fork()', 'task_kill()' and 'task_delete()' for 'Run', 'Stop' and 'Delete', respectively.

Second of all, we performed fault injection testing by applying the proposed mutation operators to RTOS. As shown in Table 4, there are two FITs. We could only modify the FIT while 'WBEND or WTEND is in execution' or 'prior to the transmission of 'ACK' or 'NAK' of the corresponding protocol'.

For the states of seven applications, we generated 56 faulty RTOS programs by applying both mutation operators, CRM and CRH to RTOS as listed in Appendix A. Applying the total number of 28 CRMs has resulted in the change of the execution path of RS-232C communication. Applying the same number of CRHs has resulted in the change of the execution path of OS API. These results represented the potential of using the faults generated by CRM and CRH when injecting faults to RTOS.

5. Conclusion and future work

RTOS, mounted on an embedded system, is responsible for running the whole system by executing applications. RTOS is time-dependent and tightly coupled with hardware devices and application. It makes RTOS difficult to test in spite of its high dependability.

The interface of RTOS and application are based on RS-232C communication protocol. The interface gets formed through *rx_symol_name* in the event of receive-communication while it gets formed through *tx_symbol_name* in the event of transmit-communication. These interfaces are the location for monitoring and debugging RTOS as well as coverage criteria for selecting test cases. We refer this as interface testing.

In this paper, we proposed the mutation operators to test fault-based interface between RTOS and application. The conventional methods to generate a mutation which is to syntactically change FIT of the original program are inappropriate in case of injecting a fault into RTOS program. As an alternative, we considered how to semantically change FIT of the programs in RTOS at a proper time in its running environment. To accomplish this, we analyzed the interfaces in three respects, 'When to inject a fault?', 'Where to inject a fault?' and 'How to inject a fault?'.

Based on the analysis of these three respects, we proposed the mutation operators CRM and CRH that change the value of *rx_mode* and *rx_header[]* respectively during the following two occasions: while 'WBEND or WTEND' is in execution and prior to the transmission of 'ACK' or 'NAK' of corresponding protocol.

We applied the mutation operators to interface testing during the integration of RTOS and application in the industrial PLC. The result from the empirical study showed the potentials of using faults generated by CRM and CRH in the fault injection to RTOS.

Currently, we focus on developing the mutation operators to test interface between RTOS and application. In the future, we plan to extend the mutation operators to test interface of different layers such as RTOS and hardware.

6. References

[1] M. A. Tsoukarellas, V. C. Gerogiannis and K. D. Economides, "Systemically Testing a Real-Time Operating System", *IEEE Micro*, Vol.15, pp.50-60, 1995.

[2] EIA/TIA-232-C, *Interface Between Data Terminal*

Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange, EIA (Electronic Industries Alliance), 1991.

[3] A. Sung, B. Choi and S. Shin, “An Interface Test Model for Hardware-dependent Software and Embedded OS API of the Embedded System”, *Journal of Computer Standards and Interfaces, ELSEVIER*, 2006, to be published.

[4] R. A. Demillo, R. J. Lipton and F. G. Sayward, “Hints on Test Data Selection: Help for the practicing programmer”, *IEEE Computer*, Vol.11, pp.34-31, 1978.

[5] A. J. Offutt, “Investigations of the Software Testing Coupling Effect”, *ACM Transactions on Software Engineering and Methodology*, Vol.1, pp.5-20, 1992.

[6] M. E. Delamaro, J. C. Maldonado and A. P. Mathur, "Interface Mutation: An Approach for Integration Testing", *IEEE Transactions on Software Engineering*, vol.27, pp228~247, 2001.

[7] Technical Report SERC-TR-41-P, *Design of Mutant Operators for the C Programming Language*, Software Engineering Research Center, Purdue University, Rev.1.04, 2006.

[8] R. T. Alexander, J. M. Bieman, S. Chosh and B. Ji., "Mutation of Java Objects", in the Proc. of International Symposium on. Software Reliability Engineering, pp.341~351, 2002.

[9] H. Yoon, and B. Choi, “Effective Test Case Selection for Component Customization and Its Application to EJB”, *The Software Testing, Verification and Reliability Journal*, vol.14, pp.45~70, 2004.

[10] A. Jerraya and W. Wolf, “Hardware/Software Interface Codesign for Embedded Systems”, *IEEE Computer*, Vol.38, pp.63~69, 2005.

[11] IEEE Standard 1003.1-2001, *IEEE Standard for Information technology – POSIX (Portable Operating System Interface)*, IEEE, 2001.

[12] J.J Labrosse, *MicroC/OS-II, The Real-Time Kernel*, CMP Books, 1999.

[13] A. Mader, “A Classification of PLC Models and Applications”, in the Proc. of International Workshop on Discrete Event Systems -- *Discrete Event Systems, Analysis and Control*, Kluwer Academic Publishers, pp.239-247, 2000.

[14] TMS320C32 Digital Signal Processor available in http://www.ti.com/, Texas Instrument, 1998.

[15] IEC, *International Standard for Programmable Controllers: Programming Languages, Technical Report IEC 1131 part 3*, IEC (International Electro technical Commission), 1993.

[16] KNICS-PLC-SDS331-01, So*ftware Design Specification for the PLC Processor Module*, KAERI (Korea Atomic Energy Research Institute), 2006.

[17] S. E. Chodrow, F. Jahnian,and M. Donner, “Run-Time Monitoring of Real-Time Systems”, in the Proc. of Run-Time Systems Symposium, IEEE, pp.74-83, 1991.

[18] S. Ricardo and Jr. J. R. de Almeida, “Run-Time Monitoring for Dependable Systems: an Approach and a Case Study”, in the Proc. of International Symposium on Reliable Distributed System, IEEE, pp.41-49, 2004.

[19] SPRU296, *Code Composer User’s Guide*, Texas Instrument, 1999.

[20] J.H. Andrews, L.C. Briand and Y. Labiche, “Is Mutation an Appropriate Tool for Testing Experiments?”, in the Proc. of International Conference on Software Engineering, pp. 402~411, 2005.

Appendix A. Generated mutants

App.	Original	Download	Run	Stop	Delete
	Mutation Operator				
App.1	Original	rx_mode=WSTX; [line # 282]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]
	CRM	rx_mode=WEOT;	rx_mode=WENQ;	rx_mode=ETB;	rx_mode=WBC0;
	Original	rx_header[1]=‘D’; [line # 275]	rx_header[1]=‘R’; [line # 311]	rx_header[1]=‘S’; [line # 311]	rx_header[1]=‘D’; [line # 311]
	CRH	rx_header[1]=‘S’;	rx_header[1]=‘C’;	rx_header[1]=‘D’;	rx_header[1]=‘R’;
App.2	Original	rx_mode=WSTX; [line # 282]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]
	CRM	rx_mode=WENQ;	rx_mode=STX;	rx_mode=SOT;	rx_mode=WBC1;
	Original	rx_header[1]=‘D’; [line # 275]	rx_header[1]=‘R’; [line # 311]	rx_header[1]=‘S’; [line # 311]	rx_header[1]=‘D’; [line # 311]
	CRH	rx_header[1]=‘S’;	rx_header[1]=‘D’;	rx_header[1]=‘C’;	rx_header[1]=‘S’;
App.3	Original	rx_mode=WSTX; [line # 282]	rx_mode=WEOT; [line # 314]	rx_mode=WEO [line # 314]	rx_mode=WEOT; [line # 314]
	CRM	rx_mode=WBEND;	rx_mode=WTB;	rx_mode=EOT;	rx_mode=WTXT;
	Original	rx_header[1]=‘D’; [line # 275]	rx_header[1]=‘R’; [line # 311]	rx_header[1]=‘S’; [line # 311]	rx_header[1]=‘D’; [line # 311]
	CRH	rx_header[1]=‘S’;	rx_header[1]=‘S’;	rx_header[1]=‘R’;	rx_header[1]=‘C’;
App.4	Original	rx_mode=WSTX; [line # 282]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]
	CRM	rx_mode=ENQ;	rx_mode=WHD0;	rx_mode=WBC1;	rx_mode=WTXT;
	Original	rx_header[1]=‘D’; [line # 275]	rx_header[1]=‘R’; [line # 311]	rx_header[1]=‘S’; [line # 311]	rx_header[1]=‘D’; [line # 311]
	CRH	rx_header[1]=‘S’;	rx_header[1]=‘C’;	rx_header[1]=‘R’;	rx_header[1]=‘S’;
App.5	Original	rx_mode=WSTX; [line # 282]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]
	CRM	rx_mode=STX;	rx_mode=WHD1;	rx_mode=WENQ;	rx_mode=EOT;
	Original	rx_header[1]=‘D’; [line # 275]	rx_header[1]=‘R’; [line # 311]	rx_header[1]=‘S’; [line # 311]	rx_header[1]=‘D’; [line # 311]
	CRH	rx_header[1]=‘S’;	rx_header[1]=‘D’;	rx_header[1]=‘R’;	rx_header[1]=‘C’;
App.6	Original	rx_mode=WSTX; [line # 282]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]
	CRM	rx_mode=WHD1;	rx_mode=EOT;	rx_mode=WTXT;	rx_mode=WHD0;
	Original	rx_header[1]=‘D’; [line # 275]	rx_header[1]=‘R’; [line # 311]	rx_header[1]=‘S’; [line # 311]	rx_header[1]=‘D’; [line # 311]
	CRH	rx_header[1]=‘S’;	rx_header[1]=‘S’;	rx_header[1]=‘C’;	rx_header[1]=‘R’;
App.7	Original	rx_mode=WSTX; [line # 282]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]	rx_mode=WEOT; [line # 314]
	CRM	rx_mode=SOH;	rx_mode=WBEND;	rx_mode=STX;	rx_mode=WHD1;
	Original	rx_header[1]=‘D’; [line # 275]	rx_header[1]=‘R’; [line # 311]	rx_header[1]=‘S’; [line # 311]	rx_header[1]=‘D’; [line # 311]
	CRH	rx_header[1]=‘S’;	rx_header[1]=‘C’;	rx_header[1]=‘C’;	rx_header[1]=‘C’;