Which Spot Should I Test for Effective Embedded Software Testing?

Jooyoung Seo, Yuhoon Ki, Byoungju Choi, Kwanghyun La†
Dept. of Computer Science and Engg., Ewha University, Seoul, Korea
Mobile Solution, System LSI Division, Samsung Electronics Co., Ltd., Yongin, Korea†
{jyseo, mirmir_}@ewhain.net, bjchoi@ewha.ac.kr, nala.la@samsung.com†

Abstract

Today, the embedded industry is changing fast – systems have become larger, more complex, and more integrated. Embedded system consists of heterogeneous layers such as hardware, HAL, device driver, OS kernel, and application. These heterogeneous layers are usually customized for special purpose hardware. Therefore, various hardware and software components of embedded system are mostly integrated together under unstable status. That is, there are more possibilities of faults in all layers unlike package software.

In this paper, we propose the embedded software interface as two essential parts: interface function *that represents the statement of communication between heterogeneous layers, and* interface variable *that represents software and/or hardware variable which are defined in different layer from integrated software and used to expected output for decision of fault. Also, we statically investigate various views of embedded software interface and demonstrate that proposed interface should be new criterion for effective embedded software testing.*

Keywords: Embedded software testing, integration testing, embedded software interface, empirical studies

1. Introduction

The embedded industry is changing fast – systems have become larger, more complex, and more integrated. Embedded software is computer software or firmware which plays an integral role in the electronics such as cars, digital televisions, airplanes, missiles, PDA, and so on. Software now becomes very sophisticated and makes up the larger part of the system, often replacing hardware [1,2]. However, embedded software occupies the 10% to 20% of embedded system, but more than 80% of faults are caused by embedded software not by hardware. Therefore, embedded software usually requires a series of rigorous testing such as structural white-box, functional black-box module and integration testing before developers release them to the market. In practice, functional testing is often more important than structural testing; also, integration testing is more challenging than module testing [3]. The main reason is that embedded software testing has the following restrictions:

Firstly, embedded software testing is often conducted under the entire system with bing-bang integration during most development cycle except unit test level. Hardware-dependency is one of the major causes of bing-bang integration testing. That is, it is very important that embedded software is loaded on hardware should be tested; because 1) interaction with hardware affects faults directly or indirectly and 2) hardware and software components of embedded system are very tightly coupled and 3) they cannot be partially executed or tested [4,5].

Secondly, because embedded system is frequently customized for dedicated purpose, software and hardware components are under development in parallel and integrated together under unstable status. Therefore, embedded software is most likely to have faults throughout entire layers compared to package software.

Thirdly, in the case of embedded software, function testing of certain software component has tendency to be performed as assuming entire system as a black-box; because software and hardware components of embedded system are tightly coupled and strongly integrated. In other words, there are no other ways to integrate and test with external software and/or hardware components without understanding their concrete behaviors. This restriction hampers precise decision where detected faults locate in: such as software under test, integrated other software components, test program, or hardware.

Accordingly, functional integration testing is very important for embedded software because abovementioned testing restriction makes it difficult to detect a fault itself and furthermore, identify its exact location and cause. Generally, functional integration testing has been defined as a testing method based on the interface which represents a function call between other layers and the function call consists of parameters and return values. In this paper, we propose the *embedded software interface* as two essential parts: '*interface function* that represents the statement of communication between heterogeneous layers', and '*interface variable* that represents software and/or hardware variable which are defined in different layer from integrated software and used to expected output for decision of fault'. Unlike package software, embedded software can occur faults depending on not only software itself but also condition of hardware. *Interface variable* plays a role as a status indicator of integrated software/hardware. Consequently, proposed embedded software interface can be new criterion for effective embedded software testing and a check-point for detecting a fault and identifying its cause.

In this paper, we elucidate why interface has to be considered as a critical criterion for embedded software testing by the following three static analysis experiments.

First of all, we investigate the premise that '*Embedded system is tightly-coupled so functional integration testing is important*'. For this, we analyze the distribution and density of embedded software interface. We measure the weight of calls between different layers among relations of total calls.

Secondly, we analyze '*Interface is more likely to occur faults*'. For this, we measure the software complexity, which is a typical metric that is widely used for fault prediction. We measure the software complexity of interface-included modules and interface-not-included modules in order to analyze how embedded software interface has an effect on software complexity. We can predict that interface-included modules are more likely to have faults through the result of analysis.

At last, we investigate '*Field faults in embedded software actually have strong relations with interface*'. For this, we analyze actual faults of the Linux-based embedded software.

The contents of this paper are as follows: Section 2 describes the embedded software interface. Sections 3 and 4 describe the experiments on which we focus, presenting research questions, measures, and experiment design [6,7]. Section 5 explains experiment results and analysis, and discussion. Finally, Section 6 describes conclusion and future work.

2. Embedded software interface

As stated in the introduction, for embedded software testing, software-software and software-hardware interface between heterogeneous layers should become a new coverage criterion and a core check-point for detecting a fault and identifying its cause. In this paper, we define embedded software interface as follows:

Definition:

Embedded Software Interface =
{ Interface function, Interface variables }

Interface function:
statement of communication between heterogeneous layers
Interface variables:
software and/or hardware variable values defined and used in other layer which is integrated with SUT (software under test)

2.1. Interface function

Interface function represents a statement of integration between SUT (software under test) and software and/or hardware in other layer. In the case of software-software integration, interface function is a statement of call between heterogeneous software layers. In the case of software-hardware integration, interface function is a statement of signal-handling form/to hardware layers.

We describe the interface function with an example of LCD device driver in Figure 1. It is embedded software that has many calls with 'Application, Kernel, Video, and Hardware' layers. Figure 1a shows LCD device driver loaded on Linux kernel v2.4.20 based on S3C2440 target processor. Of calls, the gray box in Figure 1 that calls functions in other layers means an interface function. For instance, in Figure 1b and 1c, *s3c2440fb_init_fbinfo()* in HAL calls *kmalloc()* in Kernel layer; then, *s3c244-fb_init_fbinfo()* is an interface function.

2.2. Interface variable

Interface variables are software and/or hardware variables defined in other layers integrated with SUT, and used to monitor expected output for decision of fault. Embedded software interface testing is not same as testing for identifying faults only happened at direct software-software integration. As we mentioned in introduction, it has to monitor whether integrated layers are unstable since hardware condition indirectly influence on faults in embedded software. *Interface variable* carries it out and plays the role as a status indicator.

Interface functions in following Figure 1b (*s3c2440fb_init_fbinfo*(HAL)-*kmalloc*(Kernel layer)) will give an example of what interface variables contain.

Figure 1c is the code that allocates memory for frame buffer in s3c2440 LCD driver. In package software, to verify correct execution of *kmalloc()*, what to need is only to check the function parameter of *kmalloc()* and confirm whether *fbi*, function return, is zero or not. However, embedded software is operated in restricted memory space so it can conflict with stack address even though *fbi* has meaningful address after memory allocation; and it can physically locate at boundaries of memory bank regardless of correct logical address. In these cases, software code has no problem but LCD will not operate properly because of resource restriction. In other words, return value of *kmalloc()*, *fbi*, must not be zero and there should not be any conflicts with SP (Stack Pointer) register and memory bank boundary values. Therefore, interface in this example will be as the following.

Interface function:

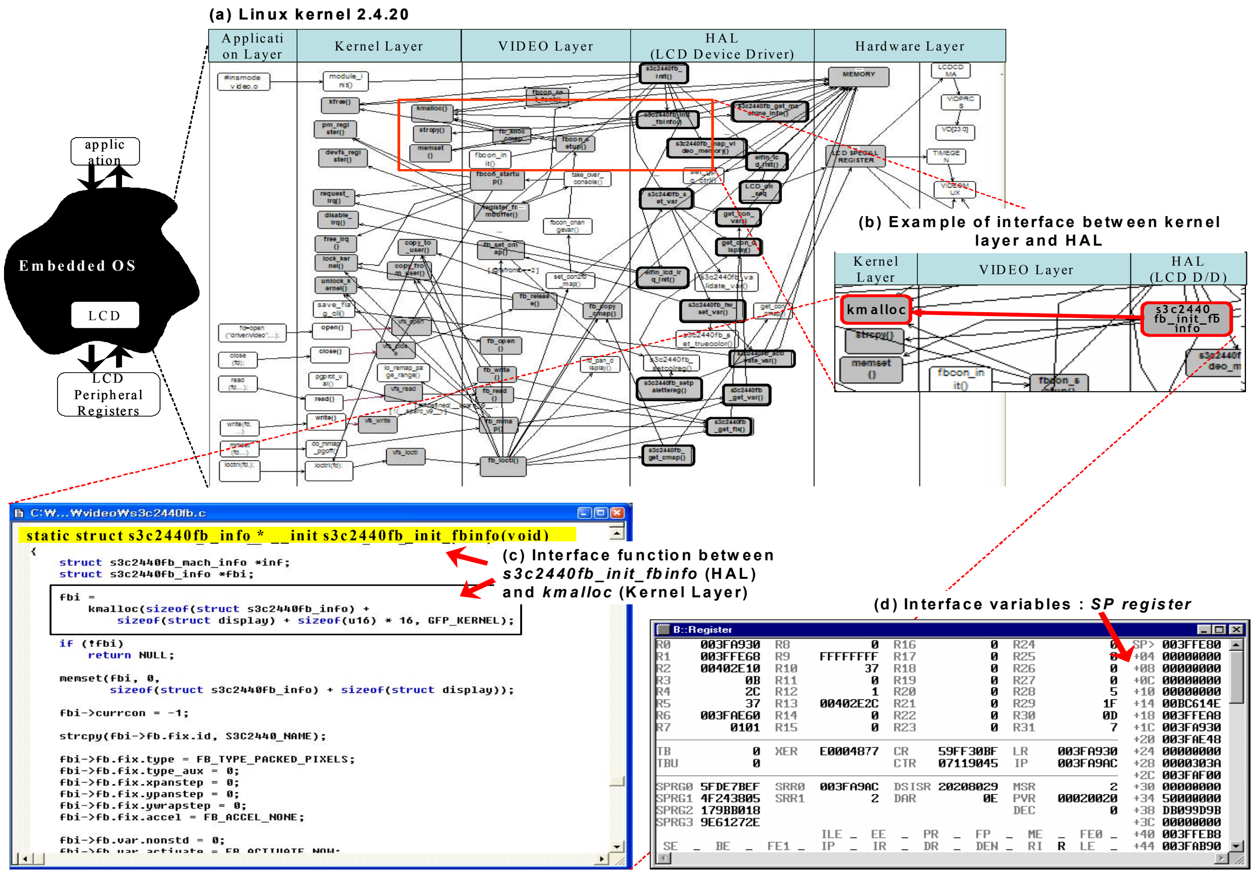
(s3c2440fb_init_fbinfo() – kmalloc())

Interface variables:

(SP register, memory bank boundary value)

We have classified embedded software interface types and published *Embedded System Interface Test Model (EmITM)*[8] that defines embedded software interface in two groups such as hardware interface and OS interface. Hardware interface is a statement including instructions to control hardware and is classified as 'memory, I/O device and timer' according to related hardware. OS interface is OS API or system call and is classified as 'task management, inter-task communication, timer management, interrupt handling, memory management, I/O management, networking and file system' according to related OS service.

This paper will elucidate that *embedded software interface* is a significant coverage criterion for embedded software testing in following chapter 3, 4 and 5 by static analysis various experiments.



```
static struct s3c2440fb_info * __init s3c2440fb_init_fbinfo(void)
{
    struct s3c2440fb_mach_info *inf;
    struct s3c2440fb_info *fbi;

    fbi =
        kmalloc(sizeof(struct s3c2440fb_info) +
            sizeof(struct display) + sizeof(u16) * 16, GFP_KERNEL);

    if (!fbi)
        return NULL;

    memset(fbi, 0,
           sizeof(struct s3c2440fb_info) + sizeof(struct display));

    fbi->currcon = -1;

    strcpy(fbi->fb.fix.id, S3C2440_NAME);

    fbi->fb.fix.type = FB_TYPE_PACKED_PIXELS;
    fbi->fb.fix.type_aux = 0;
    fbi->fb.fix.xpanstep = 0;
    fbi->fb.fix.ypanstep = 0;
    fbi->fb.fix.ywrapstep = 0;
    fbi->fb.fix.accel = FB_ACCEL_NONE;

    fbi->fb.var.nonstd = 0;
    fbi->fb.var.activate = FB_ACTIVATE_NOW;
```

```
R0     003FA930  R8             0  R16            0  R24            0  SP> 003FFE80
R1     003FFE68  R9      FFFFFFFF  R17            0  R25            0  +04 00000000
R2     00402E10  R10           37  R18            0  R26            0  +08 00000000
R3           0B  R11            0  R19            0  R27            0  +0C 00000000
R4           2C  R12            1  R20            0  R28            5  +10 00000000
R5           37  R13     00402E2C  R21            0  R29           1F  +14 00BC614E
R6     003FAE60  R14            0  R22            0  R30           0D  +18 003FFEA8
R7         0101  R15            0  R23            0  R31            7  +1C 003FA930
                                                                       +20 003FAE48
TB            0  XER     E0004877  CR      59FF30BF  LR      003FA930  +24 00000000
TBU           0                    CTR     07119045  IP      003FA9AC  +28 0000303A
                                                                       +2C 003FAF00
SPRG0  5FDE7BEF  SRR0    003FA9AC  DSISR   20208029  MSR            2  +30 00000000
SPRG1  4F243805  SRR1           2  DAR           0E  PVR     00020020  +34 50000000
SPRG2  179BB018                                      DEC            0  +38 DB099D9B
SPRG3  9E61272E                                                        +3C 00000000
                         ILE _   EE  _  PR  _  FP  _  ME  _  FE0 _     +40 003FFEB8
 SE  _  BE  _  FE1 _     IP  _   IR  _  DR  _  DEN _  RI  R  LE  _     +44 003FAB90
```

Figure 1. An example of interface in LCD device driver

3. Experiment description

3.1. Experiment definition

The overall objective of the experiment, as stated in the introduction, is to verify that the interface can serve as useful testing criterion for people who develop and test embedded software. For this, we summarize the list of research questions to be investigated.

RQ1. *"Is embedded software tightly-coupled?"*
The first step in this investigation is to analyze the manner and degree of interdependence between software and hardware components of embedded system. When it comes to difficulties in embedded software testing, there is a basic premise that the reason of difficulties is that hardware and software components are considerably tightly-coupled. In order to understand the role of interface in terms of embedded software testing, this premise should be demonstrated. For this, we measure the *interface density*, which means how much the interface accounts for in total call relations of embedded software.

RQ2. *"Do the modules that have higher interface density have the higher software complexity?"*
The second step in this investigation is to analyze software complexity and estimate probabilities of fault in the interface. We measure the *software complexity*, which is a major metric for fault prediction.

RQ3. *"Is the interface strongly involved with faults in embedded software?"*
The last step is to analyze the *relationships between interface and embedded software faults*. We then show that our interface is a key check-point of fault detection and localization. For this, we analyze actual faults of Linux-based embedded software and find out that embedded software faults are mainly occurred during integration of heterogeneous software and hardware components.

3.2. Experiment subjects

We conducted experiments with software in various layers of embedded system to reveal importance of interface in embedded software testing. Eleven C programs were used as subjects (see Table 1). The programs range in size from 186 to 6001 lines and cover a variety of embedded system layers and target processor.

Table 1. Experiment subjects

Program	Layer	Target Processor	Size : LOC (Capacity*)	# of Calls	# of Interfaces
P_1 Wave Player	Embedded Application	S3C2440	3568	391	173
P_2 Camera**	Embedded Application	S3C2440	334	90	73
P_3 Embedded Linux	Embedded OS	S3C6400	(22.4MB)*	28863	17260
P_4 LCD	System Software	S3C2443	4586	435	192
P_5 RTC	System Software	S3C2443	827	114	70
P_6 TOUCH SCREEN	System Software	S3C2443	495	76	41
P_7 UART	System Software	S3C2443	6001	698	220
P_8 IIS	System Software	S3C2443	1013	210	132
P_9 NAND	System Software	S3C2443	3265	416	110
P_{10} MMC	System Software	S3C2443	4747	278	91
P_{11} CAMERA***	System Software	S3C2440	186	46	46

(** Application, *** Device Driver)

4. Experiment design

To address our first question (RQ1), we need to measure the number of interface in heterogeneous layers. *Interface density* is equal to degree of calls in different layers over all calls. Therefore, we measure Interface Size / Total Call Relation Size.

Question 2 can be addressed by analyzing the *software complexity*. Software complexity of the embedded software is measured in two parts such as interface-included modules and other modules. Through the result of this measurement, we can find out how much the interface have impacts on software complexity. For this, we use a metric, *McCabe's Module Design Complexity* (iv(G)) [10] and do measurement by using McCabe IQ test tool [11].

We investigate question RQ3 by analyzing the relationships between the embedded software interface and fault. For this, we see the number of faults in the interface between heterogeneous layers over total faults on the basis of actual faults of the Linux-based embedded software [13].

The experiment manipulates six independent variables:

- Measure 1: Total Call Relation Size (N_c)
- Measure 2: Interface Size (N_i)
- Measure 3: Module Design Complexity (iv(G))
- Measure 4: Interface Function Size (N_f)
- Measure 5: Total Detected Faults Size (F_t)
- Measure 6: Detected Faults Size in Interface (F_i)

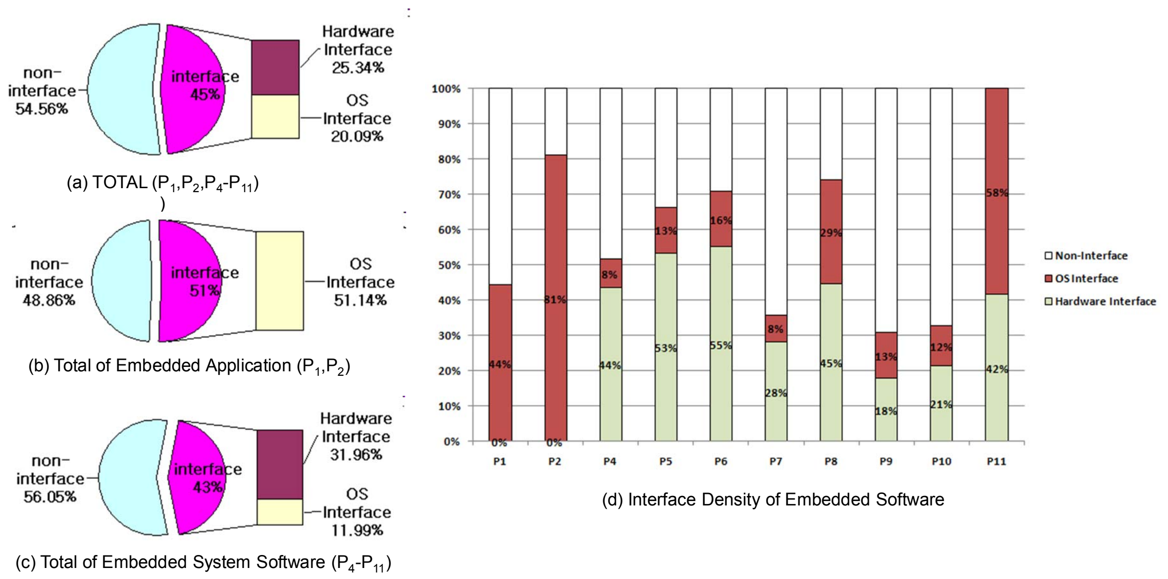


Figure 2. Interface density of embedded software

Six dependent variables are measured:

- Interface Density (%): $(N_i * 100) / N_c$
- Interface Complexity: $\sum iv(G)_i / N_f$
- Interface Fault (%): $(F_i * 100) / F_t$

5. Analysis and results

5.1. Interface density

We examined interface distribution in Figure 2 as analysis result of diverse embedded software such as such as P_1, P_2 (application layer), and P_4-P_{11} (system software layer) in Table 1. In Figure 2a, interface occupies 45% of total calls; hardware interface is 25.34% and OS interface is 20.09%.

(a) High proportion of OS interface

In application software (Figure 2b), OS interface occupies pretty big proportion, 51.14%. It seems that subject program P_1 and P_2 have influence on the results because they operate on OS. However, the general belief that OS-based operation is more stable than others cannot be applied to embedded software; because, unlike OS for package software, embedded OS such as 'MS WinCE, Symbian, and Embedded Linux' is implemented frequently ported with device drivers and architecture-dependency codes for changed target processor. Therefore, OS interfaces (51.14%) in results are not stabilized interaction and it is possible to be indirectly influenced since there are potential faults in codes frequently ported due to hardware-dependency.

(b) High proportion of hardware interface

In system software (Figure 2c), hardware interface is 31.96%. System software like device driver is implemented for the purpose of controlling certain hardware, so it has very close interaction with Hardware layer and wrong interaction with hardware readily results in faults.

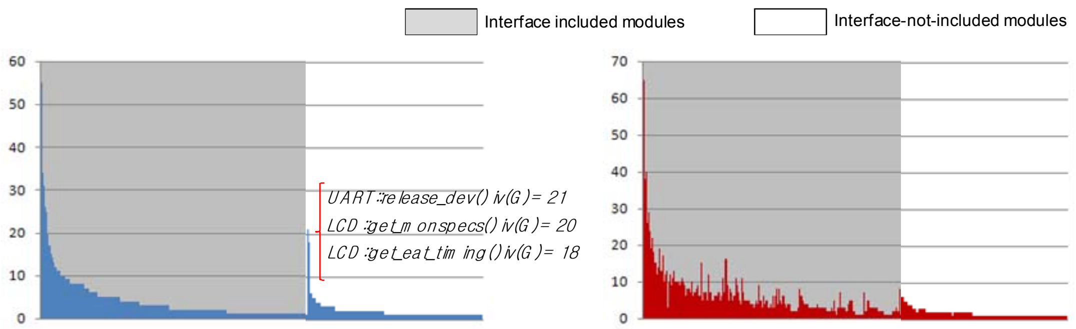
5.2. Interface complexity

Module design complexity (iv(G)) reflects the complexity of the module's calling patterns to its immediate subordinate modules. This metric differentiates modules which will seriously complicate the design and program from modules which simply contain complex computational logic [13]. That is, iv(G) is an important metric to show dependency between software modules and fault prediction. In order to investigate software complexity and estimate a probability of fault in the interface, we measured iv(G) of system software (Table 1) and Figure 3 is the result.

In Figure 3, each bar of x axis represents each module and y axis represents iv(G) value. X-axis is sorted by iv(G) grouped by *interface-included* module and *interface-not-included* module. As a result, on the average interface-included module is more complicated than interface-not-included module and its module dependency is stronger.

(a) High value in the part of *interface-not-included*

In Figure 3a, there are few modules with high iv(G) value in the part that doesn't include interface. For



(a) McCabe's module design complexity of embedded software:
Interface-included: AVG(iv(G) = 3.652
Interface-not-included: AVG(iv(G)) = 1.719

(b) Refined McCabe's module design complexity of embedded software:
Interface-included: AVG(iv(G) = 5.431
Interface-not-included: AVG(iv(G)) = 1.579

Figure 3. Interface complexity of embedded software

example, iv(G) for UART *release_dev()* is 21 and iv(G) is 20 and 18 respectively for *get_monspecs()* and *get_eat_timing()*; the values are relatively high compared to other values in the part of *interface-not-included*. But most calls of these modules are print functions for debugging. There functions are not usually related to a fault so complexity is measured higher than real value.

(b) Low value in the part of *interface-included*

In Figure 3a, there are some modules with iv(G) value 1 in the part with interface. iv(G) value 1 means modules with sequence statements without any calls or with single call. It is possible that only one call is interface. However, according to our analysis, most of modules consist of sequence statements without calls when its iv(G) value is 1. Also, most of modules with value 1 are in HAL and consist of statements that have hardware interface to control hardware. Generally, a statement to control hardware in embedded software is essential code that highly impacts on the faults so complexity without considering this is measured lower than real value.

(c) Refined software complexity for the embedded software interface

iv(G) measures +1 for function call. We adjusted the interface value to +2 reckoning that implement becomes more complicated and possibility of faults becomes higher when using interface in unknown area compared to using known internal function calls. In addition, hardware interface is not implemented by function calls but by hardware control statements so it is not much reflected on iv(G). Thus, we adjusted iv(G) value for more precise measurement of complexity for embedded software: in terms of fault detection, we took account of over-estimated or under-estimated case and this is displayed in Figure 3b.

- Hardware control statement (H/W interface): +2
- Interface among heterogeneous layers: +2
- Print function call for debugging: +0

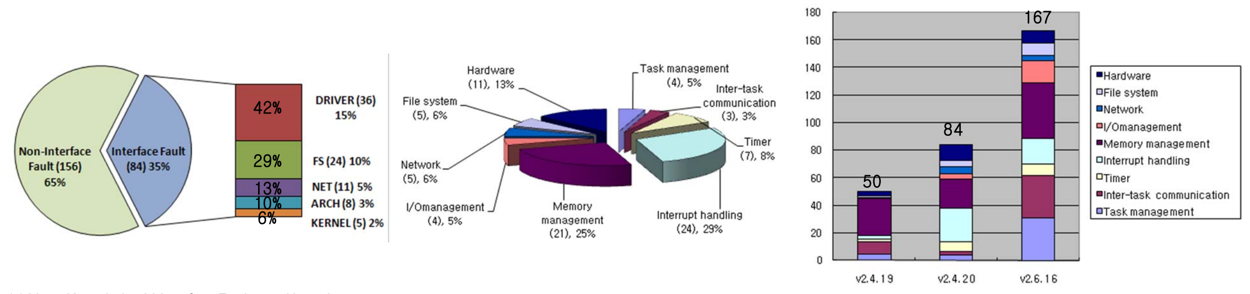
As a result, through adjusted iv(G) value, *interface-included* module has 5.431 of iv(G) on the average and relatively higher complexity than *interface-not-included* which has 1.579.

5.3. Interface fault

We analyzed the relation between embedded software interface and fault. It is based on change-log of Linux-based embedded software [12,14]. A subject of experiment, change-log of P_3, consists of 1498 logs such as 'version change, compiler makefile change, configuration change, compile warning modification, fault fix, patch, requirement extension and target hardware extension'. The log for pure fault is 16.02% (240 faults) of total 1498 logs. We divided more as Figure 4 according to location and cause of fault.

(a) Interface fault location

We classified actual faults as interface fault and non-interface fault according to fault location. As a result, interface faults are 35.0% (84 faults) of all as Figure 4a. This proves previous two experiments demonstrate 'the possibility of interface fault in embedded software is relatively high'. That is, it shows that there are actually many faults in interface between heterogeneous layers. Especially, Figure 4a shows that 42% of faults happen in interface of device driver and it is because device driver is tightly-coupled at both Kernel layer and Hardware layer.



(a) Linux Kernel v2.4.20 Interface Faults and Location (b) Linux Kernel v2.4.20 (c) Linux Kernel v2.4.19-v2.6.16 Interface Faults

Figure 4. Distribution of interface fault location and cause

(b) Interface fault cause

We analyzed cause of interface faults and non-interface faults, separately. First of all, as the result of analyzing fault cause in non-interface, major cause was programming mistakes such as 'wrong type, wrong indentation, wrong calculation and condition scope error'. In addition, analyzing fault cause in interface according to EmITM[8], it displays that 29% of faults are at interrupt handling, 25% are memory management and 13% are hardware as Figure 4b. This trend is same for various Linux versions in Figure 4c.

Consequently, when fault location is not interface, these fault causes are corresponded to simple bugs that can be debugged by unit test. However, many interface faults were associated with the functional integration faults and unable to be detected unless test target is integrated in entire system.

5.4. Discussion

Three experiment results discovers that proposed interface is the critical check-point for detecting and identifying faults and should be a coverage criterion of embedded software testing as the following.

Firstly, we figured out that portion of interface between heterogeneous layers is very high through analyzing *interface density* of embedded software. Also, faults are likely to happen throughout all integrated layers compared to package software. Possibility of faults in interface, furthermore, becomes much bigger because software is tightly-coupled around interface as seen in the result of conducted experiments.

Secondly, analysis of *software complexity* demonstrated that interface-included module is much more complicated than module with normal calls. Complexity is a well known metric used for fault prediction and the result showed that interface-included module is more likely to contain faults.

These two results represent that embedded software is very tightly-coupled and possibility of faults is high based on the interface. Tightly-coupled and high-integrated characteristics of embedded software makes it difficult to detect faults and it is demanding to localize faults and exam the cause even after faults are discovered. Hence, interface, the part where software is integrated, should be test coverage criteria of embedded software and this is the hot-spot for detecting and debugging faults.

At third, we analyzed the actual faults of Linux-based embedded software and found out that faults frequently happen when heterogeneous components with different characteristics and features are integrated. Figure 5 displays an actual fault that was found in Intel sound card driver on Linux kernel.

```
■ Sample: Interface fault related the Task Management
  /* /sound/pci/intel8x0.c:Fix sleep in atomic during prepare callback */
  if (cnt & ICH_PCM_246_MASK) {
        iputword(chip, ICHREG(GLOB_CNT),
                   cnt & ~ICH_PCM_246_MASK);
+       spin_unlock_irq(&chip->reg_lock); // Fix code for unloking IRQ
        msleep(50);
+       spin_lock_irq(&chip->reg_lock);    // Fix code for locking IRQ
  }
```

Figure 5. An example of interface fault

This example is one of parts in Intel sound card driver pulse conversion code from channel 2 to channel 6. Pulse conversion is implemented by calling *iputword()*, callback function, and putting a device in sleep condition during preparation. When *msleep()* for task management in kernel layer is under execution, *spin_unlock_irq()* and *spic_lock_irq()* should be implemented in pairs in order not to lock IRQ during then; however, this code cannot be tested without integrating all together such as Device driver layer, Kernel layer and Hardware layer.

We have been testing Linux-based embedded software of *A* Co., Ltd., applying interface-based test criterion and we could detect above- mentioned faults a lot [9]. For example, the fault in Figure 5 could be found by *interface function* checking the location of *msleep()* and by *interface variable* confirming status of IRQ unlock.

Consequently, we could discover why proposed interface should be the coverage criterion of embedded software testing and why this would be the hot-spot for fault detection. Therefore, we can define the embedded software test coverage as follows:

Embedded Software Test Coverage *requires every Embedded Software Interface in a program to be executed at least once, where*

Embedded Software Interface =
{ Interface function, Interface variables }

6. Conclusion and future work

In this paper, we proposed that embedded software interface is the location of call between heterogeneous layers and, at the same time, a variable value of software and/or hardware which is defined and used in different layer from integrated software. Also, we statically analyzed *interface density*, *software complexity* and *relationship of interface and fault* for various embedded software and demonstrated that proposed interface should be new criterion of embedded software for test case selection and test coverage analysis.

Currently, we have defined the embedded software interface test method on the basis of results of experiments. Along with this, we have developed an automated testing tool for embedded software, *Justitia*[9,15], which includes four automated core test activities such as test case selection, test driver generation, test execution, and test coverage analysis. We then have applied *Justitia* to Linux-based mobile platform testing in *A* Co., Ltd. We have been experimenting to measure test coverage, fault efficiency and accuracy in order to show the effectiveness of proposed interface-based test method.

In future work, we plan to conduct comparative experiment with white-box test coverage such as control-flow or data-flow test so as to elucidate that interface-based test coverage is more effective in terms of fault detection capability.

7. Acknowledgements

This work was supported in part by the Korea Research Foundation Grant funded by the Korean Government(MOEHRD)(KRF-2006-531-D00027).
This work was supported in part by the Samsung Electronics Co., Ltd., 2005-2007. We received the Great Enterprise S/W Innovation Award from Samsung Electronics Co., Ltd., 2007.9.

8. References

[1] Bart, B., Edwin, N., Testing Embedded Software, Addison-Wesley, 2003.
[2] Jerraya A.A, Wolf, W., “Hardware/software interface codesign for embedded systems”, IEEE Computer, 2005, pp.75-84.
[3] Wei-Tek, T., Lian, T., Feng, Z., Ray, P., “Rapid embedded system testing using verification patterns”, IEEE Software, 2005, pp68-75.
[4] Yoo S., Jerraya A.A., “Introduction to Hardware Abstraction Layers for SoC”, in the Proc. Of Design, Automation and Test in Europe Conf. and Exhibition (DATE), IEEE, 2003, pp.10,336-10,347.
[5] Lee, E.A, “What’s ahead for embedded software?”, IEEE Computer, 2000, pp.18-26.
[6] Barbara A. K., Shari L. P., David C. H., Jarrett R., “Preliminary Guidelines for Empirical Research in Software Engineering”, IEEE Transactions on Software Engineering, Vol. 28, No.8, 2002, 721-734.
[7] Lj. Lazic, D. Velasevic, “Applying simulation and design of experiments to the embedded software testing process”, Software Testing, Verification and Reliability, Vol.14, 2004, pp257-282.
[8] Sung, A., Choi, B., Sin, S., “An interface test model for hardware-dependent software and embedded OS API of embedded system”, Computer Standard & Interface, Vol.29, 2007, pp430-443.
[9] *A* Co., Ltd., Technical Report of Test Model and Automated Test Tool (Justitia) for Mobile AP Device Driver, 2007.
[10] Arthur, W., Tom, M., NIST document – “Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric”, available in http://www.mccabe.com/iq_research_metrics.htm
[11] McCabe’s IQ Test Tool, available in http://www.mccabe.com/iq.htm
[12] Linux kernel Change Log, available in http://www.kernel.org/pub/linux/kernel/
[13] Henry, Kaufura, Harris, “On the relationships among three software metrics”, ACM Workshop/Symposium on Measurement and Evaluation of software quality, 1981, pp81-88.
[14] Linux kernel Change Log, available in http://linux.bkbits.net:8080
[15] Seo, J., Sung, A., Choi, B., Kang, S., “Automated embedded software testing on an emulated target board”, ICSE Workshop on Automation of Software Test, USA, 2007.