Automating Embedded Software Testing on an Emulated Target Board

Jooyoung Seo, Ahyoung Sung, Byoungju Choi, Sungbong Kang[a]
Dept. of Computer Science and Engg., Ewha University, Seoul, Korea
Mobile Solution, System LSI Division, Samsung Electronics Co., Ltd., Yongin, Korea[a]
{jyseo, aysung}@ewhain.net, bjchoi@ewha.ac.kr, osbkang@samsung.com[a]

Abstract

An embedded system consists of heterogeneous layers including hardware, HAL (Hardware Abstraction Layer), OS kernel and application layer. Interactions between these layers are the software interfaces to be tested in an embedded system. The identified interfaces are important criterion that selects test cases and monitors the test results in order to detect faults and trace their causes. In this paper, we propose an automated scheme of embedded software interface test based on the emulated target board. The automated scheme enables to identify the location of interface in the source code to be tested, to generate test cases, and to determine 'pass' or 'fail' on the interface. We implemented the test tool called 'Justitia' based on the proposed scheme. As a case study, we applied the 'Justitia' to mobile embedded software on the S3C2440 microprocessor and Linux kernel v2.4.20.

1. Introduction

Lately, the presence of the Ubiquitous, 'Embedded, Everywhere' from the cell phones to D-TV and DMB phones, makes the embedded systems widespread rapidly on real life [1,2]. Embedded software occupies the 10~20% of embedded systems. More than 80% of the system faults are caused by not hardware but embedded software. Considering the trend of increase in embedded software, the embedded software problem is no longer considered as a simple software fault. Therefore, so the embedded software test becomes very important.

Furthermore, the importance of software in the embedded system's market has an impact on testing. Time-to-market and quality become major competing factors. It means that more and more complex software has to be developed at a high quality level in less time. A well-structured test process alone is not sufficient to fulfill the quality demands within stipulated time. With the proper use of test tool, the faster testing with good quality is possible [3].

Recent trend shows that embedded software is tested via a debugger equipped from the simulator/emulator or a test tool to measure the performance of entire embedded system [4]. Each of the case has some limitations to test embedded software.

First, in order to detect a fault, the debugger allows software engineers to set a break point, determine a symbol to be monitored, and decide 'pass' or 'fail' of a test result. If software engineers are not quite experienced in testing or they do not know architecture of the entire embedded system, they cannot execute the ad-hoc testing by trial and error. They take more time to test, and they cannot confirm whether the test was completed reliably and the fault evaluation has been performed properly.

Second, in order to solve the hardware dependency for embedded software testing, the test tool should support the test environment such as a simulator, an emulator, and a real target [5]. The test tool focuses to automate the test execution based on the test environment. A good test tool should support the labor-intensive tasks such as the test execution, and the technology-intensive tasks including test item identification, test case selection, and test result analysis oriented to embedded software. In order to help the software engineer, a test tool should support above mentioned tasks.

Many test tools have limitations to analyze and test the internal features of embedded software that is tightly coupled with OS, middleware, device driver and target hardware. To analyze the interfaces between the heterogeneous layers is very important to the embedded software testing. As it is impossible to test embedded software independently, it is tested in the entire embedded system. Testing the entire embedded system is difficult to detect the potential software fault,

Second International Workshop on Automation of Software Test (AST'07)
0-7695-2971-2/07 $20.00 © 2007 IEEE

its location and cause. Therefore, the interface test of embedded software is important to identify the fault location.

In this paper, we propose an automated scheme of embedded software test based on the emulated target board. The proposed scheme is based on two techniques as follows:

First, we propose *interface test technique* between embedded system layers to generate test case. The proposed interface test technique is the system integration test focused on the interface which is overlapped on each layer for various kinds of embedded products. We define the interface pattern of embedded software and maps the pattern to test feature of *EmITM* (Embedded System's Interface Test Model) [6] which defines the interface test feature in general for embedded system.

Second, we propose *emulation test technique* to execute test case. The proposed emulation test technique is combination of the monitoring and debugging techniques of the existing emulator debugger with testing embedded software. That is, the proposed technique makes full use of setting the breakpoints on the interface, and monitoring symbol to determine 'pass' or 'fail' of the test results.

We automate these techniques and develop the test tool, called *Justitia*. As a case study, we apply the *Justitia* to mobile software and demonstrate the test results.

The contents of this paper are as follows: Section 2 and Section 3 describe the embedded software interface test and its automation scheme. Section 4 explains the test tool, *Justitia*, and the case study of its application. Finally, Section 5 describes conclusion and further study.

2. Embedded software interface test

2.1. Embedded software interface

The embedded system is composed of different layers, such as hardware layer, HAL, OS layer and embedded application layer [2,7] as shown in Figure 1. Because software and hardware units at every layer are tightly coupled and strongly correlated, it is impossible to independently execute and test embedded software itself. Even if we want to test specific software such as device driver only, we have no choice but to test entire embedded system.

In this paper, we focus on embedded software interface testing between OS layer and Hardware layer. We classify various software and hardware units in each layer, SUk, SUd, SUp, HUd and HUi to identify the focal point in embedded software testing as follows:

SUk. SUk is kernel dependent software which is an OS service based on system calls and API.

SUd. SUd is software to support physical device, such as LCD, USB, touch screen and UART.

SUp. SUp is processor-dependent software in HAL which initializes and configures the target hardware.

HUd. HUd is a RAM or Register which is controlled with software units directly.

HUi. HUi is a device, such as UART, USB, and Watchdog timer which is controlled with software units through HUd indirectly.

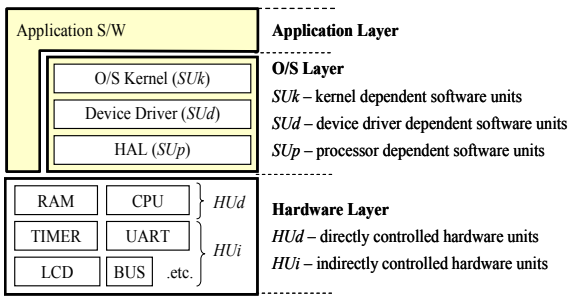

Figure 1. Layers and units of embedded system

The gateway that is in charge of communications between two different layers is the focal point in embedded software testing. In this paper, we define this gateway as 'interface'. Taking s3c2440fb_init_fbinfo() in Figure 2 as an example, kmalloc(), memset(), and strcpy() represent the interface between SUp and SUk. The identified interfaces are important criterion that selects test cases and monitors the test results in order to detect faults and trace their causes.

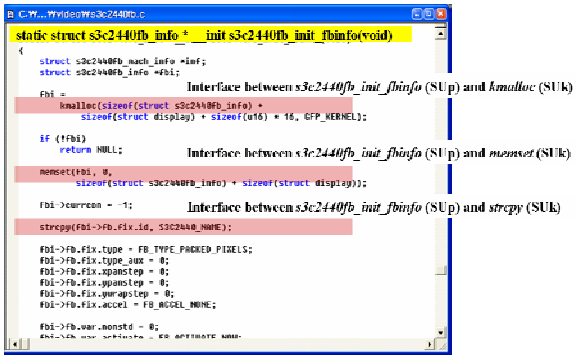

Figure 2. An example of interface between SUp and SUk

2.2. Embedded software interface test

0-7695-2971-2/07 $20.00 © 2007 IEEE

There are many interfaces in an embedded system. In order to identify test features for each interface, we have classified two kinds of interface called HPI (Hardware Part Interface) and OPI (OS Part Interface) and developed the Embedded System Interface Test Model called *EmITM*. HPI is the interface that directly accesses the hardware and OPI is the interface using OS services. The *EmITM* defines the test features for both HPI and OPI. The test features of HPI are extracted from the hardware design specifications. They consist of total 15 sub-test features regarding memory, I/O device and a timer. The test features of OPI are composed of total 106 sub-test features related to process management, inter-process communications, time management, interrupt/exception, memory management, I/O management, networking, and file system as shown in Table 1. The capital letters in Table 1 represent software or hardware units including interface, such as SUk and HUd. These units decide the interface test feature. In the case of HPI, HUd is MEMORY, SPECIAL REGISTER OF I/O DEVICE or SPECIAL REGISTER OF TIMER. In the case of OPI, SUk is the OS service based on POSIX1003.4 [7], ELCPS [8] and KELPS [9].

Table 1. *EmITM*

Interface	Interface Test Features	
HPI	Memory	Memory allocation, Memory access: MEMORY
	I/O Device	I/O operation: SPECIAL REGISTER OF I/O DEVICE
	Timer	Timer controller register setup, Timing parameter setup Response time: SPECIAL REGISTER OF TIMER
OPI	Process Management	Process create, context-switch, terminate: _JOB_CONTROL, _SINGLE_PROCESS, _SPAWN, _THREADS_REALTIME, _PRIORITY_SCHEDULING, _THREADS_EXT, _THREAD_PROCESS_SHARED, _THREADS, HREAD_PRIORITY_SCHEDULING, _BARRIORES, _THEADS_REALTIME_EXT, _SPIN_LOCKS) Multi-threading and multi-processing: _MULTI_ADDR_SPACE, _MULTI_PROCESS Scheduling: _SCHEDULING
	Inter-Process Communication	Synchronizing: _SEMAPHORES, _PIPE Inter-process communication: _IPC, _SEMAPHORES, _PIPE
	Time management	Time delay: _TIMERS Time read/write: _CLOCK_SELECTION
	Interrupt and Exception Handling	Context recovery and signal handling: _SIGNALSI5, _REALTIME_SIGNALS, _SIGNAL_JUMP
	Memory Management	Memory allocation, Memory collision, Memory leakage, Page fault, Critical region: _DYNAMIC_LINKING, _MEM_MGMT, _MAPPED_FILES, _MEMORY_PROTECTION, _MEM_LOCK, _MEM_LOCK_RANGE, _MULTI_ADDR_SPACE, _REG_EXP
	IO Management	Device driver: _ASYNCHRONOUS_IO, _DEVICE_SPECIFIC_R, _DEMULTI_ADDR_SPACE, _WIDE_CHAR_DEVICE_IO _DEVICE_SPECIFIC Device I/O request handling: _DEVICE_IO
	Networking	Networking data structure, Networking system calls, Packet send/receive: _NETWORKING Remote procedure call: _NETWORKING_RPC
	File System	Disk management, File I/O: _FD_MGMT, _SYNCHRONOZIED_IO, _FYNC _FIFO, _FILE_ATTRIBUTES, _FILE_SYSTEM, FILE_SYSTEM_EXTEN_FILE_SYSTEM_R, _LARGE_FILE _STDIO_LOCKING

Interface pattern. In order to test the interfaces of embedded software, we first identify the interfaces and their test features as shown in Table 1. We automate this process by analyzing the executed control flow between SUk, SUd, SUp, HUd and HUi. As a result, we define five kinds of interface patterns as shown in Figure 3. According to the executed control flow of HUd, we define two patterns of HPI such as HPI_1 and HPI_2:

HPI_1. SUd or SUp directly accesses the HUd, such as a CPU register and a memory to read or write the value.

HPI_2. SUd or SUp indirectly controls HUi such as a timer and a LED through the address of HUd

According to OS services (SUk), we define three patterns of OPI such as OPI_1, OPI_2, and OPI_3:

OPI_1. SUd or SUp calls SUk that is unrelated to the control of HUd and HUi. SUk is the system calls or APIs related to process management, inter-process communications, and exception handling.

OPI_2. SUd or SUp calls SUk that is indirectly related to the control of HUd, such as virtual memory. SUk is the system calls or APIs related to virtual memory management.

OPI_3. SUd or SUp calls SUk that is directly related to the control of HUd and HUi. SUk is the system calls or APIs related to physical memory management, time management, interrupt handling, I/O management, networking, and file systems.

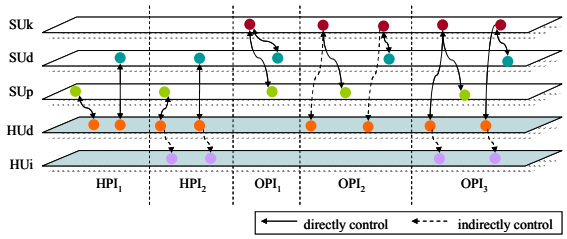

Figure 3. Interface patterns of embedded software

The test coverage of embedded software is based on the interfaces which are extracted according to the interface patterns described in Figure 3. These interfaces are the locations of the symbol to generate test cases and to decide 'pass' or 'fail' of the test results. Also, they become the monitoring locations to trace the causes of faults during a debugging process.

3. Automated scheme of embedded software interface test

Test activities consist of test plan, test design, test execution, and test result analysis. Especially, the test design and execution are repeated for many times to find out and recover the faults. It is difficult to perform the repetitive test activities without automation.

Mostly, the testing automation for embedded software focuses on the test execution including debugging and performance measurement. If the automation of designing test case is not supported, software testing is limited to monitor unspecific memory and CPU registers. In addition, it would be difficult to identify the specific location where the faults are densely populated. In order to help the

0-7695-2970-9/07 $20.00 © 2007 IEEE

software engineer, it is essential to automate not only test execution but also test design.

In this section, we propose the test scheme for embedded software to automate the generation and execution of test cases, and analysis of test results as shown in Figure 4. The major idea of proposed scheme is integration between the debugging/monitoring techniques of emulator and the interface testing described in Section 2. In other words, our scheme enables automatically to identify the interface, to generate the test cases covered the interface test, and to execute the test cases on the emulator. In addition, debugging information enables to detect a fault and trace its causes automatically.

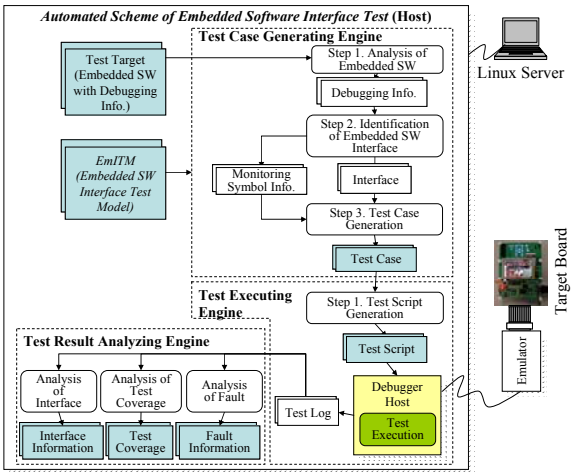

Figure 4. Automated scheme of embedded software interface test

The proposed scheme consists of the following 3 engines: 1) *test case generating engine*, 2) *test executing engine*, and 3) *test result analyzing engine*. We describe the *test case generating engine* and the *test executing engine* in Section 3.1 and 3.2, respectively. And we describe the *test result analyzing engine* in Section 4 with the case study.

3.1. Test case generating engine

The *test case generating engine* generates test cases by analyzing the executable image of embedded software. **The test case includes the 'interface test feature, location of interface, interface symbol to be monitored at the interface, input data and expected output' as follows**:

Interface test feature. It means the test item to be tested at the interface. We have defined the interface test feature in *EmITM*. The *test case generating engine* extracts the location of interface and decides the interface symbol, input data and its expected output from the interface test feature.

Location of interface. It means the address of interface and the software unit including interface. The *test executing engine* sets a breakpoint on the location of interface to detect faults.

Interface symbol. It means the input and return parameter of interface. The *test executing engine* monitors value of the interface symbol to determine 'pass' or 'fail'. The interface symbol is a different variable in the interface. According to the procedure call standard for the target processor [10], it is restricted to the general registers, such as R0, R1, R2 and R3. The interface symbol is one of these general registers instead of a specific variable.

Input data & Expected output. They mean the input data and its expected output of the interface symbol. Especially, the expected output is a return value of software or hardware unit including the interface. We decide the expected output according to the hardware design specification and OS API standards.

Figure 5 shows the architecture of embedded Linux kernel which consists of memory manager, device manager, file system manager, networking manager and device drivers. Because the entire Linux kernel is too complex to describe the inter-relationship of different layers, we take the LCD device driver to show the process of getting a grip of the interface as an example. Gray box in Figure 5 represents the software and hardware which are tightly coupled with LCD device driver. Figure 6 represents the architecture of LCD device driver in detail.

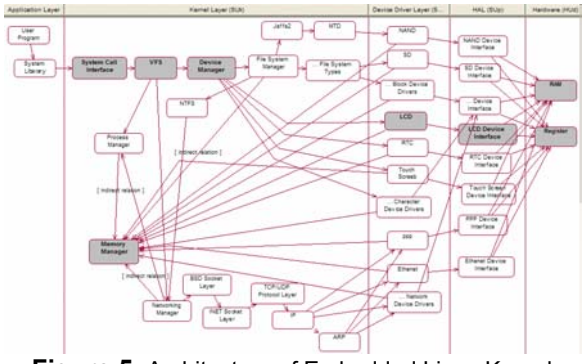

Figure 5. Architecture of Embedded Linux Kernel

Step 1 Analysis of embedded software. It is necessary to analyze embedded software for identifying the interface. The proposed scheme has full advantage of debugging information such as symbol table and disassembly code. We analyze the symbol table to identify interface symbols. We also analyze the

0-7695-2970-9/07 $20.00 © 2007 IEEE

disassembly code to find out caller-callee relations that are used to identify the interface in Step 2.

Step 2 Identification of embedded software interface. The details of identifying interface are as follows:

- To extract caller-callee relations.
- To classify the software and hardware units belonging to each embedded system layer as described in Section 2.1.
- To extract units that include interfaces; According to interface pattern in Section 2.2, interfaces are identified between embedded system layers.

The box in Figure 6 show caller-callee relation and the gray box show software unit including interface, respectively.

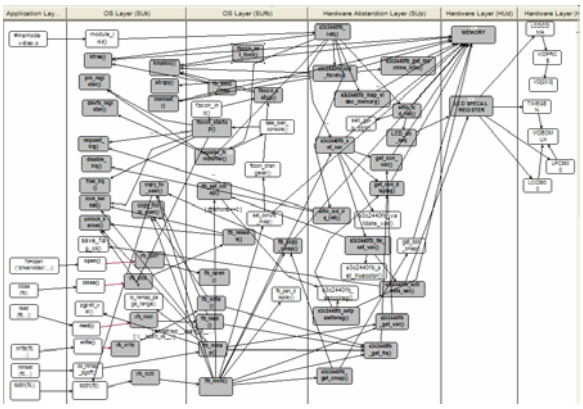

Figure 6. Interfaces of Linux LCD device driver

Step 3 Test case generation. The test case generating engine automates the mapping between interface test features in Table 1 and interfaces. Taking HPI as an example, the engine selects the HUd that includes the interfaces between the layers. Taking OPI as an example, the engine selects the SUk that includes the interfaces between the layers.

Table 2 shows a sample test case of Linux LCD device driver for *Memory Allocation* test feature in Table 1. The interface location for *Memory Allocation* is decided according to SUk, such as kmalloc() and malloc(). In Figure 6, kmalloc() is coupled with fbcon_set_font(), fbcon_setup(), fb_alloc_cmap() and s3c2440fb_init_fbinfo(). Taking kmalloc() as an example, the interface symbols are the input parameter of kmalloc() and the monitoring location that contains the return value of kmalloc(). As described in Section 3.1, the monitoring location is represented as general registers like R0, R1 and R2. R0 means size parameter of kmalloc() and R1 means flag. Taking TC1 as an example, the expected output is a return value of kmalloc() which should be greater than 0 immediately after the memory allocation. The *test case generating engine* monitors the expected output with R0.

Table 2. A sample test case of Linux LCD device driver for *Memory Allocation* test feature

Test Case	Interface Test Feature	Interface Location	Interface Symbol	Input Data	Expected Output
TC1	Memory Allocation	0xC001303C, s3c2440fb_init_fbinfo()	R0	R0 : 0x32C R1 : GFP_KERNEL	R0 > 0
TC2	Memory Allocation	0xC00DB998, fb_alloc_cmap()	R0	R0 : 0x1E0 R1 : GFP_KERNEL	R0 > 0
TC3	Memory Allocation	0xC00DC8C0, fbcon_setup()	R0	R0 : 0x100 R1 : GFP_KERNEL	R0 > 0
TC4	Memory Allocation	0xC00DFE6C, fbcon_set_font()	R0	R0 : 0x1D0 R1 : GFP_KERNEL	R0 > 0

3.2. Test executing engine

In order to execute test cases automatically, we propose *emulation test technique* that is a combination of monitoring/debugging techniques and interface testing. That is, the proposed technique makes full use of setting the breakpoints on the interface, and monitoring symbol to determine 'pass' or 'fail' as shown in Figure 7.

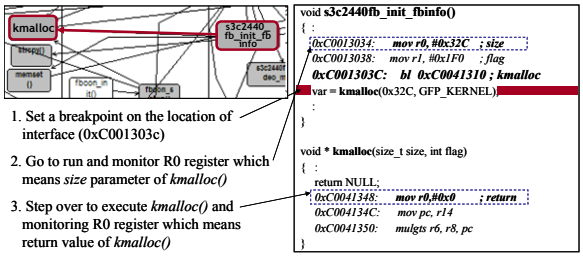

Figure 7. Monitoring and debugging steps for TC1 in Table 2

Step 1 Test script generation. The *test executing engine* generates test case with a script language. The **test script executes test cases on an emulator automatically and consists of various debugging commands** as follows:

Breakpoint. It is a command for setting breakpoints on the interfaces.

Go & Step. It is a command for running the target board.

Monitoring. It is a command for monitoring the current value of interface symbol.

IF…ELSE. It is a command for comparing the current value of interface symbol with expected output and deciding 'pass' or 'fail'.

Print. It is a command for saving the test log.

Figure 8 shows the test scripts for TC1.

0-7695-2970-9/07 $20.00 © 2007 IEEE

```
// A script for setting a breakpoint on the location of interface
Break.Set 0xC001303C
LOOP:
    // A script for running target board
    Go
    wait !run()
    // A script for monitoring symbol
    Local &size &var &expected_output &pc_addr
    &pc_addr=R(PC)
    &expected_output=0
    &size=R(R0)
    Step.Over
    &var=R(R0)
    // A script for comparing current value with expected output
    // and deciding ' pass'or ' fail'
    IF &var>&expected_output
    (       // A script for saving the test log – Pass
        Print "$$ PASS"
        Print "Location: &pc_addr &file-#&line"
        Print "Allocation Info: Start Addr - &var  Size - &size"
    ) ELSE ( // A script for saving the test log – Fail
        Print "$$ FAIL"
        Print "Fault: NOT_MEMORY_ALLOC_FAULT"
        Print "Fault Location: &pc_addr &file #&line"
    )
    Goto LOOP
// Delete breakpoints
Break.deleteall
```

Figure 8. A sample of test script for TC1

4. Justitia and its Case study

4.1. Justitia

We have developed the *Justitia v1.0* based on the test scheme proposed in section 3. The *Justitia* was implemented in C/C++ under Windows XP operating system. The *Justitia* is a tool for testing embedded software mounted on the Linux kernel v2.4.20. The Linux for S3C2440 RISC microprocessor is an embedded OS providing the device drivers, such as UART, USB, camera, I2C, and MMC for Mobile AP (Access Point). It is applicable to a PDA phone and a Smartphone including MMI, MMS, WAP Browser, Email, Java applications and phone controls. The *Justitia* adopts the TRACE32-ICD (JTAG emulator), which supports debugging features such as breakpoint, run, and symbol monitoring. TRACE32-ICD provides the PRACTICE script language to run automatic test.

4.2. A Case study

We tested the embedded software mounted on Linux kernel for S3C2440 microprocessor using the *Justitia*. We summarize our case study in the interface test coverage and interface faults.

4.2.1. Interface test coverage. The *Justitia* provides the interface test coverage, calculated as interfaces executed to interfaces extracted. As shown in Table 3, we have found 21 faults when testing 337,410 set of test cases among 342,128 interfaces extracted by the *Justitia*.

Table 3. Interfaces coverage

Interface Pattern	Interface Test Feature		# of Interface	# of Covered Interface	Interface Coverage	# of Fault
HPI_1	Memory	Memory allocation	3277	3277	100.0 %	0
		Memory collision	3277	3277	100.0 %	0
OPI_2	Memory Management	Memory allocation	239	16	6.7 %	0
		Memory collision	1827	329	18.0 %	4
		Memory leakage	418	49	11.7 %	7
OPI_3		Miss alignment	329463	329463	100.0 %	8
		Critical region	2603	7	0.3 %	0
OPI_3	Interrupt	Context recovery	17	1	6.0 %	1
		Interrupt service	17	1	6.0 %	1
Total			342128	337410	98.6 %	21

As an example of interface test coverage, Figure 9 shows that *Memory Collision* test feature is to be tested in fbcon_show_logo() interface and it is tested fully as 100% coverage. As a result, the *Justitia* shows the number of passed and failed test cases.

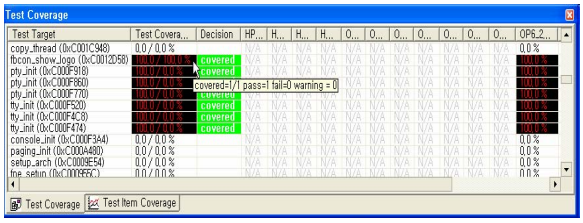

Figure 9. Test coverage screen

4.2.1. Interface faults. As shown in Table 4, we found 21 faults; 2 faults were detected by *Interrupt* and 19 faults were detected by and *Memory Management*.

Table 4. Interface faults

Interface Test Feature	Interface Location / File #Line	Fault Type	Interface Layer
Context recovery	__irq_svc / irq.c #40	Context recovery fail	SU_k and HU_d
Interrupt handling	elfine_timer_interrupt() / time.c #30	Interrupt service routine disable	SU_k, HU_d and HU_i
Memory collision	padzero () / uaccess.h-#221	Heap collision	SU_k and HU_d
	create_elf_tables / uaccess.h-#214	Heap collision	SU_k and HU_d
	init_std_data() / random.c-#1426	Heap collision	SU_k and HU_d
	get_zeroed_page() /page_alloc.c-#45	Heap collision	SU_k and HU_d
Memory leakage	vc_resize() / console.c-#771	Zero address space free	SU_d, SU_k, and HU_d
	kfree_skbmem() / skbuff.c-#302	Zero address space free	SU_k and HU_d
	load_elf_binary() / binfmt_elf.c-#473	Memory leakage	SU_k and HU_d
	load_elf_binary() / binfmt_elf.c-#506	Memory leakage	SU_k and HU_d
	s3c2440fb_set_var() / s3c2440fb.c-#412	Memory leakage	SU_d, SU_k, and HU_d
	vc_resize() / console.c-#721	Memory leakage	SU_d, SU_k, and HU_d
	pipe_new() / pipe.c-#447	Memory leakage	SU_k and HU_d
Page fault	linux_logo_green / 0xC0017267	Data miss alignment	SU_k and HU_d
	linux_logo_blue / 0xC0017322	Data miss alignment	SU_k and HU_d
	linux_logo / 0xC00173DD	Data miss alignment	SU_k and HU_d
	linux_logo_bw / 0xC0018CDD	Data miss alignment	SU_k and HU_d
	linux_logo16 / 0xC0018FFD	Data miss alignment	SU_k and HU_d
	cpu_elf_name / 0xC0022DC2	Data miss alignment	SU_k and HU_d
	usb11_rh_dev_descriptor / 0xC0151316	Data miss alignment	SU_k and HU_d
	hs_rh_config_descriptor / 0xC0151341	Data miss alignment	SU_k and HU_d

Figure 10 (a) shows results of *Memory Collision*. It shows conflict symbol, such as name, address, size,

0-7695-2970-9/07 $20.00 © 2007 IEEE

and location of interface which consists of file name and line number. Here, the 'location' is useful in jumping to the exact source line where a fault has occurred. For example, there were heap collision faults at the system_utsname_domainname[59] and system_utsname+0x187. At this moment, the interface is init_std_data() that is located in the line no.1426 of /home/aysung/s3c2440_kernel/drivers/char/random.c.

Figure 10 (b) shows results of *Context Recovery*. The context recovery was failed at the IRQ 0x3D in do_IRO(); 0x3D means the interrupt of touch screen based on the S3C2440 specification.

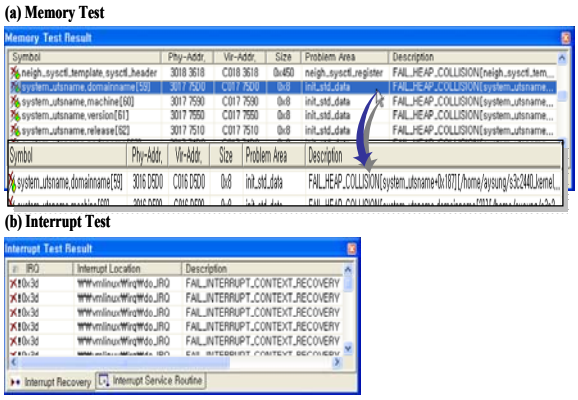

Figure 10. Test result screen

5. Conclusion

We proposed a scheme of embedded software testing in the emulation test environment before proceeding to mount on a real hardware. The key contributions in this paper are as follows:

First, we proposed the interface test for embedded software. Because an embedded system has a hierarchical structure with multiple layers, we focused on the interface where software and hardware units are tightly coupled.

Second, we automated to identify interfaces to be tested and to generate test cases that covered the interface. Our test cases include 'interface test feature, location of interface, symbol to be monitored at the interface, input data, and expected output'.

Third, we proposed the *emulation test technique* to execute test case. The *emulation test technique* lies in the integration of the interface test on the emulator with debugging and monitoring functions. That is, the proposed technique makes full use of setting the breakpoints on the interface and monitoring the symbols to determine 'pass' or 'fail'. Generally, the software engineer cannot just execute the developed embedded software independently. Therefore, they perform the test and debug with the entire system that was merged with individual code. The software engineer cannot find out whether the fault evaluation has been conducted properly or not and where the exact location of the fault is occurred. However, we resolved hardship by integrating the test tasks, such as fault detection and tracing its cause, to the simple debugging and monitoring functions in existing emulators.

We developed the *Justitia v1.0*, a new embedded software testing tool and applied the tool to test the embedded software mounted on the Linux kernel v2.4.20 and the S3C2440 microprocessor for PDA phone and Smartphone. The *Justitia v1.0* support '*Memory, Interrupt, Memory Management*' test features of *EmITM*.

Currently, we are developing the *Justitia v2.0* to support all the test features in Table 1. In the future, we plan to extend interfaces of different layers, such as embedded application and middleware.

6. Acknowledgements

This work was supported in part by the Samsung Electronics Co., Ltd. This work was supported in part by the Korea Research Foundation Grant funded by the Korean Government(MOEHRD)(KRF-2006-531-D00027).

7. References

[1] Abowd, G.D., "Software engineering issues for ubiquitous computing", *Proc. of Intl' Conference on Software Engineering (ICSE)'99 (21th)*, 1999, pp.75-84.

[2] Jerraya, A.A., Wolf, W., "Hardware/software interface codesign for embedded systems", *IEEE Computer*, 2005. pp. 63-69.

[3] Graaf, B., Lormans, M., Toetenel, H., "Embedded software engineering: The state of the practice", *IEEE Software*, 2003. pp.61-69.

[4] VDC (Venture Development Corporation), Embedded Software Market Intelligence Program – Embedded Software Test Automation Tools, 2004.

[5] Bart, B., Edwin, N., Testing Embedded Software, Addison-Wesley, 2003.

[6] Sung, A., Choi, B., Sin, S., "An interface test model for hardware-dependent software and embedded OS API of embedded system", *Computer Standard & Interface*, 2006.

[7] IEEE Std. 1003.1-2001: *IEEE Standard for Information Technology–POSIX,* 2001.

[8] ELC (Embedded Linux Consortium), *ELC Platform Specification v1.0.* 2002.

[9] KESIC (Korea Embedded Software Industry Council), *KESIC Embedded Linux Platform Specification v.1.0*, 2004.

[10] Earnshaw, R., *ARM Procedure Call Standard for the ARM Architecture*, 2005.

0-7695-2970-9/07 $20.00 © 2007 IEEE