Automatic Generation of Hardware/Software Interface with Product-Specific Debugging Tools*

Jeong-Han Yun†, Gunwoo Kim†, Choonho Son‡, and Taisook Han†

†Dept. of Computer Science, Korea Advanced Institute of Science and Technology
‡Network Technology Laboratory, Korea Telecom
{dolgam,reshout}@pllab.kaist.ac.kr, choonho@kt.co.kr, han@cs.kaist.ac.kr

Abstract. Software programmers want to manage pure software, not hardware-software entanglements. Unfortunately, traditional development methodologies cannot clearly separate hardware and software in embedded system development process. We propose a *Hardware/software INterface GEnerator*; we call it HINGE. After receiving device specifications including device usage rules for each device, HINGE automatically generates device API, device driver, and device controller for each device. In addition, HINGE equips device APIs to check the device usage rules at run-time. Consequently, HINGE gives support to not only fast prototyping but also device usage rule-debugging in embedded software.

1 Introduction

Traditional embedded system development process repeats requirement analysis, separated hardware/software design, implementation, and test. This process raises three controversial arguments. First, this process needs many implementation-iterations caused by information miscommunication or requirement changes. Any of them impacts the entire development team. Second, hardware/software interface(Figure 1) implementation and integration are tedious and error-prone; 40% of product development time is spent for system integration, and more 60% of operating system's errors are born from here[1]. Finally, system debugging and requirement check are too hard. Equivocal error messages from device drivers do not tell the exact cause of errors.

To overcome these problems, we propose a *Hardware/software INterface GEnerator*; we call it HINGE. After receiving an *interface specification* for each device from system designers, HINGE generates the hardware/software interface for each device.

Moreover, HINGE generates rule-checking codes based on device usage rules in API specification, and inserts them into device APIs. The inserted codes

*This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031)

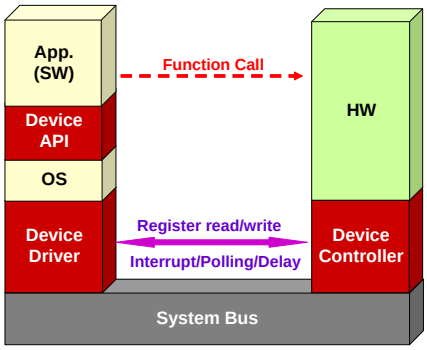

Fig. 1. Hardware/Software Interface: device API, device driver, and device controller

notice wrong usages of devices using return values of device APIs[1]. With the generated device APIs, software-development groups can easily detect device usage errors by the return values of the device APIs at run-time. Therefore, all of the generated device APIs is suitable for a good rule-debugging tool or a device exception handler.

This paper is organized as follows. Section 2 defines the interface specification as HINGE's input. Section 3 explains the automatic generation of hardware/software interface. The case study of HINGE is in Section 4. Section 5 summaries the related work about hardware/software interface. Section 6 concludes the paper.

2 Interface Specification

Interface specification is our representation of hardware/software interface information as HINGE's input. It is divided into four categories as follows:

API Specification : declaration of interface relations
Driver Specification : definition of communication events[2]
Controller Specification : memory assignment of hardware
System Specification : information of a target machine

Each specification is written in XML format.

2.1 API Specification

Push-Pull interface has been used to show who is a major actor between sender and receiver. We apply Push-Pull interface to *API specification.*

[1] Alarms with special return values generally apply to software libraries; for example, the return value 0 of `malloc` function in C libraries means the failure of dynamic memory allocation.

[2] The events can be interpreted as variables in software, and as signals in hardware.

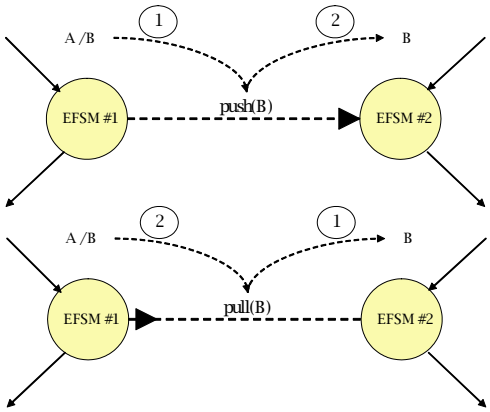

Fig. 2. Push interface and Pull interface: our graphic notation

Push-Pull Interface We can represent communications between two EFSMs[6] by Push-Pull interface. Push interface means that a receiver always detects[3] events as soon as a sender emits[4] it. On the other hand, Pull interface means that a receiver detects events when it wants. Using these two interfaces, we can represent delivery/processing order of communication events exactly.

Both Push interface and Pull interface in Figure 2 are delivering an output event B from EFSM #1 as an input event to EFSM #2, but the behaviors between EFSM #1 and EFSM #2 are different.

From API specification, When an event is occurred, EFSM #1 sends the event into EFSM #2 by Push mechanism. That is, EFSM #1 is an active sender, and EFSM #2 is a passive receiver. Therefore, the sender controls the synchronization timing of two EFSMs in Push interface. Push interface can be manipulated by interrupt or polling mechanism.

However, EFSM #2 decides the timing or receiving events in Pull interface, even if the event is occurred in the EFSM #1. That is, EFSM #1 is a passive sender, and EFSM #2 is an active receiver. Therefore, the receiver controls the synchronization timing of two EFSMs in Pull interface.

API specification API specification consists of three parts:

1. Function type of each device API
2. Interface definition using Push-Pull interface
3. Automata for describing device usage rules: the execution order among device APIs or permitted range of parameters

[3]Detecting events can be considered as reading the data of variables with regard to software and as receiving signals with regard to hardware

[4]Emitting events can be interpreted as writing the data of variables in respect of software and as sending signals in respect of hardware.

```
<Driver>
  <event name='adder'>
    <sw2hw sw_var='uint8 a' hw_port='A [7:0]' />
    <sw2hw sw_var='uint8 b' hw_port='B [7:0]' />
    <hw2sw sw_var='uint8 c' hw_port='C [7:0]' />
    <interrupt pin_name='27' hw_name='ready' />
  </event>

  <event_layout id='adder' virtual_address='0xf1810000'>
    <offset id='0' sw='a' map='[7:0]' to='A [7:0]'/>
    <offset id='1' sw='b' map='[7:0]' to='B [7:0]'/>
    <offset id='2' sw='c' map='[7:0]' from='C [7:0]'/>
  </event_layout>
</Driver>
```

Fig. 3. An example of driver specification

HINGE generates device API from API specification. Whenever the device API is called, it checks the device usage rules. That is, it returns positive value that indicates the current state number on device usage rule automata. If embedded software breaks an device usage rule, the device API returns negative one of the current state number. We will show an example in Section 4.

2.2 Driver Specification

Driver specification is used by HINGE to generate device drivers. Figure 3 is an example. Driver specification is composed of two parts:

1. communication event : the connection between the device driver's arguments and hardware ports
2. communication event layout : the mapping to virtual address

HINGE uses memory mapped I/O for hardware/software communication[5]. So all events must be mapped to their own fixed virtual addresses in kernel memory.

2.3 Controller Specification

Controller specification contains low-level information such as system bus and the memory space of hardware. Figure 4 is an example of controller specification. Using these information, HINGE generates device controller: a composition of address decoder and data decoder.

[5]In our experimental environment, devices are implemented on FPGA for prototyping. FPGA module has physically continuous address space. This address space is used for kernel memory.

```
<Controller>
  <AddrDecoder>
    <addr id='CX_A' width='[21:1]'/>
    <chip_select id='NPX_CS5' type='active low'/>
    <write from='0' to='1'/>
    <read from='2' to='2'/>
  </AddrDecoder>
  <DataDecoder>
    <data id='CX_D' width='[15:0]'/>
    <write_enable id='NPX_PWE' type='active high'/>
    <read_enable id='GPIO27' type='active low'/>
  </DataDecoder>
</Controller>
```

Fig. 4. An example of controller specification

$$
\begin{aligned}
\mathit{SYSTEM} &::= \mathit{CPU}\ \ \mathit{OS} \\
\mathit{CPU} &::= \mathit{CPUTYPE}\ \ \mathit{ENDIAN} \\
\mathit{CPUTYPE} &::= \mathit{xscale} \mid \mathit{x86} \\
\mathit{ENDIAN} &::= \mathit{little} \mid \mathit{big} \\
\mathit{OS} &::= \mathit{LINUX} \\
\mathit{LINUX} &::= \mathit{KERNEL}\ \ \mathit{DRIVER} \\
\mathit{KERNEL} &::= \mathit{2.4} \mid \mathit{2.6} \\
\mathit{DRIVER} &::= -1 \mid \mathit{PositiveInteger}
\end{aligned}
$$

Fig. 5. The abstract grammar for system specification

2.4 System Specification

System specification has the information about CPU and operating system. Figure 5 shows the abstract grammar of system specification.

Now HINGE supports Linux operating system with Intel Xscale architecture. The kernel version of Linux is one of 2.4 or 2.6, which are the major versions of the Linux kernel. The `DRIVER` specifies a major number of hardware devices. If the specified number is a positive integer, the device is statically allocated by that number. Otherwise, the device is dynamically allocated by the operating system.

3 Hardware/Software Interface

From the interface specification, HINGE automatically generates hardware/software interface: device API, device driver, and device controller. Figure 6 is our target board architecture.

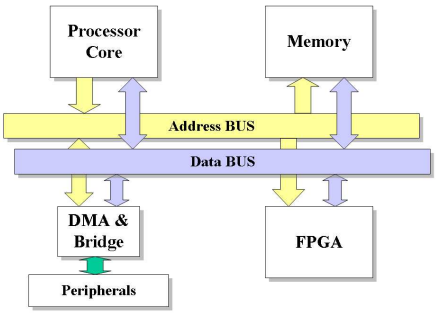

Fig. 6. Our target board architecture

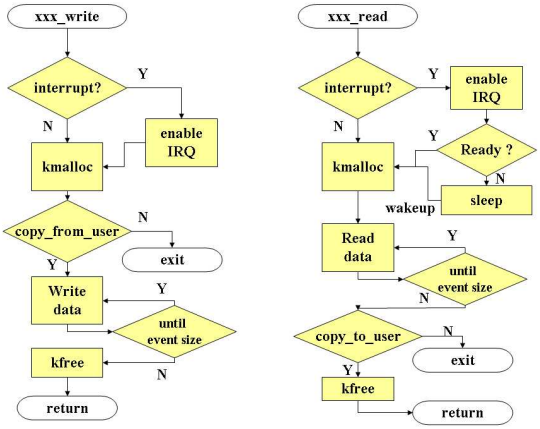

Fig. 7. The models of `write` and `read` functions

3.1 Device API

Device API is a high-level encapsulation of device drivers; embedded software can access devices like software libraries through device API.

Especially, device API generated by HINGE provides debugging faculties which can detect wrong usages of device APIs with the following procedure:

1. Each device API memorizes the device's current state in its device usage rule automata.
2. If device API is called in a wrong way, it returns the negative value of the current state number.
3. If device API is called in a right way, it moves to the next state by this usage and returns the positive value of the state number.

After checking above conditions, the device API transfers hardware/software communication events using the device drivers.

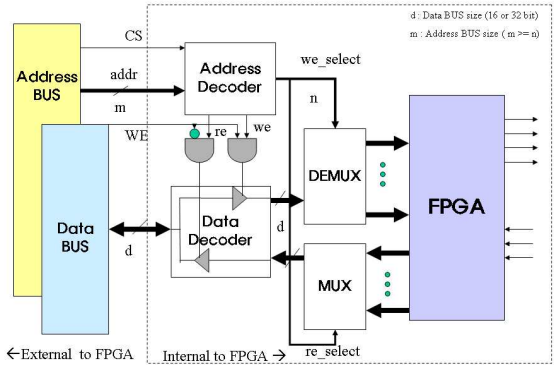

Fig. 8. Schematic diagram of the read/write module

3.2 Device Driver

HINGE generates character device drivers for target operating systems. Fundamental functions are `open`, `close`, `read`, `write`, and `llseek`.

In Linux environment, each function of device driver has common model. HINGE generates device drivers through these common device driver models. These models can be applied to any device driver. Figure 7 shows the models of `write` and `read` functions.

Like `write` and `read` functions, HINGE automatically generates other device driver functions based on the common driver models.

3.3 Device Controller

The device controller consists of an address decoder and a data decoder. The address decoder gets a physical address from the address bus, and emits control signals for the data decoder. The data decoder reads or writes data according to control signals emitted by the address decoder.

A device can be classified into three functional models by behaviors of the data decoder: read module, write module, and read/write module. Among of them, Figure 8 shows the schematic diagram of the read/write module. HINGE generates Verilog codes based on that schematic diagram.

4 Case Study

We will show the advantages of HINGE with the case of a simplified automatic transmission. For the experiment, we use EMPOS II[2] with embedded Linux(kernel version 2.4). The devices are written in Verilog HDL. The devices are loaded on FPGA with the generated device controllers, and the device drivers are loaded on embedded Linux.

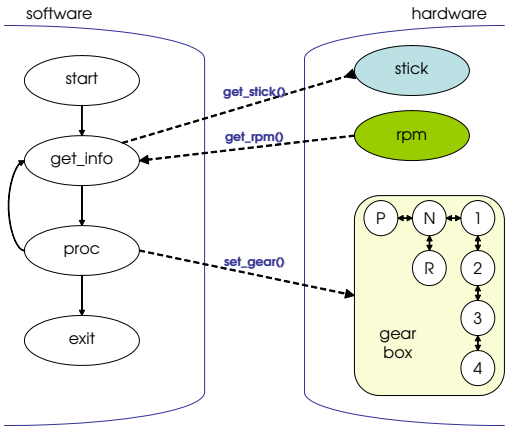

Fig. 9. Push-Pull Interface of simplified automatic transmission

4.1 Design of Automatic Transmission System

The software gets the current lever position from the lever device by Pull interface, and the rpm gauge tells the software of the current rpm by Push interface. After deciding appropriate gear ratios based on the lever position and the current rpm, the software orders the gearbox device to change gear ratios by Push interface. Figure 9 is the informal representation of the entire system using our graphic notation for Push-Pull interface.

4.2 The Use of HINGE

To check whether the software uses devices correctly, we should express device usage rules by state automata in API specification. In case of the gearbox device, it has several states corresponding to ratios. Its API specification is shown in Figure 10. A ratio -1 denotes the reverse gear, and a ratio 100 does the parking gear; others are omitted.

The experimental results of simplified automatic transmission are depicted in Table 1 and Table 2.

By using generated device APIs instead of kernel functions, software can check or handle error cases at run-time. For example, when the current gear ratio is set to 'second gear', if the software orders the gearbox device to set gear ratio 'neutral gear', `set_ratios` returns '-SECOND'; '-SECOND' indicates the error occurrence on the device state 'SECOND' in the device usage rules(Figure 11). Each state name automatically defines a constant in generated device APIs.

5 Related Work

Codesign Tools The major difference between HINGE and interface generators of other codesign systems is the device API to check the device usage rules at

```
<API>
  <device id='gearbox'>
    <func name='set_ratios'>
      <param type = 'uint8' name='ratios'   size='1'    op='write' />
    </func>
  </device>
  <automata>
    <state name='NEUTRAL'>
      <func name='set_ratios'>
        <action cond='ratios == 0'  to='NEUTRAL' />
        <action cond='ratios == -1' to='REVERSE' />
        <action cond='ratios == 1'   to='FIRST' />
      </func>
    </state>
    ...
    <state name='SECOND'>
      <func name='set_ratios'>
        <action cond='ratios == 1' to='FIRST' />
        <action cond='ratios == 3' to='THIRD' />
      <func>
    </state>
    ...
    <state name='PARKING'>
      <func name='set_ratios'>
        <action cond='ratios == 0'   to='NEUTRAL' />
        <action cond='ratios == 100' to='PARKING' />
      </func>
    </state>
  </automata>
</API>
```

Fig. 10. A portion of the API specification of the hardware component gearbox

run-time. To the best of our knowledge, none of codesign frameworks service the debugging interface like that.

Approaches to automatic interface generation including CHINOOK[5] and POLIS[6] describe a system with a set of codesign finite state machines(CFSMs) which use FIFO queues. Since these codesign environments keep all of target architecture information, development groups using these environments do not write the hardware/software interface specification. This is why these environments are not flexible for architecture modification.

COSMOS[7], COSYMA[8], and CoWare[9] use concurrently-running processes using remote procedure calls(RPCs) for communications. This mechanism is more complicate than shared memory and may cause inefficiency. COSMOS and CoWare are now commercially available from AREXSYS[10] and CoWare[11].

```
void auto_transmission() {
  int chk, lever, rpm;
  ...
  while(1) {
    chk = get_lever(&lever);
    if (chk < 0) {
      printf("get_lever error : error_code=%d\n", chk);
      exit(1);
    }
    chk = get_rpm(&rpm);
    if (chk < 0) {
      printf("get_rpm error : error_code=%d\n", chk);
      exit(1);
    }
    if (lever==0) {
      chk = set_ratios(0);
      if (chk== -SECOND) { // if current gear ratio is second
        if (rpm < 1500) {
          chk = set_ratio(1);
        }
      }
      else if (chk < 0) {
        printf("set_ratios error : error_code=%d\n", chk);
        exit(1);
      }
    }
    ...
  }
}
```

Fig. 11. Software `auto_transmission` using device APIs

Interface Generation Traditionally, device drivers have been written in C due to its efficiency and flexibility. Unfortunately, sophisticated device interaction protocols and C's lack of type safety make driver code complex and prone to failure. Device drivers account for 70–90% of bugs in the Linux kernel and have error rates up to three to seven times higher than the rest of the kernel[12]. So various approaches have been suggested to improve the reliability of low-level software, device driver software, and device controller in many researches[1, 4, 13–19].

These tools support device interfaces well. Nevertheless, bugs of embedded software may be born from wrong device usages as well as device interfaces implementation itself. Some of them provide fundamental but simple debugging interfaces[1, 4, 9, 19]. However, our tool generates device APIs that can alarm wrong device usages based on API specification. This helps embedded software debugging and exception handling for wrong device usages.

	API	Driver	Controller	System	Total
`stick`	14	8	12	9	43
`rpm`	14	9	12	9	44
`gearbox`	56	8	12	9	85
`Total`	84	25	36	27	172

Table 1. Automatic_transmission's interface specification(lines of code)

	API (C)	Driver (C)	Controller (Verilog)	Makefile	Total
`stick`	33	87	74	15	209
`rpm`	33	94	74	15	216
`gearbox`	68	88	92	15	263
`Total`	134	269	240	45	688

Table 2. Generated codes from the interface specification in Table 1(lines of code)

6 Conclusion

Our ultimate goal is the separation of hardware- and software-development process. For that purpose, we propose interface specification as hardware/software interface description methodology and implement the hardware/software interface generator HINGE. Our approach makes four contributions to embedded software development.

First, API specification using Push-Pull interface can show the more realistic interfacing mechanisms than other interface description methodologies.

Second, interface specification can be used for a common specification document between hardware- and software-development groups. Especially, software development groups can learn the device usage rules for product requirements.

Third, HINGE automatically generates all of hardware/software interfaces for target devices. Automatic hardware/software interface generation can significantly reduce the burdens of embedded system development.

Finally, generated device APIs return negative values when embedded software violates the device usage rules. So, the device APIs generated by HINGE can help requirement-, function- or rule-level tests; software programmers can omit additional work just like rule-check function programming.

References

1. QuickDriver, http://www.etri.re.kr/www_05/search/view_03.php?fclass=01&idx=1247
2. Hanback Electronics CO.LTD, http://www.hanback.co.kr/
3. A. Rajawat, M.Balakrishnan, and A. Kumar, "Interface Synthesis : Issues and Approaches", *Proceedings of the 13th International Conference on VLSI Design*, pp.92–97, 2000.

4. WindRiver, http://www.windriver.com/
5. P. Chou, R. Ortega, and G. Borriello, "Interface co-synthesis techniques for embedded systems", *Proceedings of the IEEE/ACM International Conference on CAD (ICCAD)*, pp.280–287, 1995.
6. F. Balarin, A. Jurecska, and H. Hsieh et al, *Hardware-Software Co-Design of Embedded System: The Polis Approach*, Kluwer Academic Press, Boston, 1997.
7. T. B. Ismail, M. Abid and A. Jerraya, "COSMOS: A codesign approah for communicating systems", *Proceedings of the 3rd International workshop on Hardware/software Co-Design*, pp.17–24, Grenoble, France, 1994.
8. R. Ernst, J. Henkel, T. Benner, W. Ye, U. Holtmann, D. Hermann, and M. Trawny, "COSYMA environment for hardware/software cosynthesis of small embedded systems", *Microprocessors and Microsystems*, Vol.20, pp.159–166, 1996.
9. D. Verkest, K. Van Rompaey, and I. Boolsens, *Co-Ware - A Design Environment for Heterogeneous Hardware/Software Systems*, 1, Nov. 1996.
10. Arexsys. http://www.arexsys.org/
11. CoWare. http://www.coware.com/
12. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors", *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, volume 35 of *Operating System Review*, pp 73–88, Banff, Alberta, Canada, October 2001.
13. F. Hessel, P. Coste, P. Lemarrec, N. Zergainoh, J. M. Daveau, and A. A. Jerraya, "Communication and Interface Synthesis on a Rapid Prototyping Hardware/Software Codesign System", *IEEE International Workshop on Rapid System Prototyping*, 1998.
14. M. Eisenring and J. Teich, "Domain-Specific Interface Generation from Dataflow Specifications", *Proceedings of the 6th International Workshop on Hardware Software Codesign*, pp. 43–47, 1998.
15. L. Palopoli II, G. Lipari, L. Abeni, M. D. Natale, P. Ancilotti, and F. Conticelli, "A Tool for Simulation and Fast Prototyping of Embedded Control Systems", *Languages, Compilers, and Tools for Embedded Systems*, pp. 73–81, 2001.
16. F. Merillon, L. Reveillere, C. Consel, R. Marlet, and G. Muller, "Devil: An IDL for Hardware Programming", *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, pp. 17-30, San Diego, California, October 2000.
17. S. Wang and S. Malik, "Synthesizing Operating System Based Device Drivers in Embedded Systems", *Proceedings of the First International Conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS)*, Newport Beach, CA, October 2003.
18. S. A. Edwards, "SHIM: A language for Hardware/Software Integration", *Synchronous Languages, Applications, and Programming*, 2005.
19. C. L. Conway, and S. A. Edwards, "NDL: A Domain-Specific Language for Device Drivers", *Languages, Compilers, and Tools for Embedded Systems*, pp. 30–36, Washington, DC, June 2004.