Automatic Hardware/Software Interface Generation for Embedded System

Choonho Son*, Jeong-Han Yun, Hyun-Goo Kang**, and Taisook Han****

Abstract - A large portion of the embedded system development process involves the integration of hardware and software. Unfortunately, communication across the hardware/software boundary is tedious and error-prone to create. This paper presents an automatic hardware/software interface generation system. As the front-end of hardware/software co-design frameworks, a system designer defines XML specifications for hardware functions. Our system generates hardware/software interfaces including *Device Driver, Driver API*, and *Device Controller* from these specifications. Embedded software designers can easily use hardware just like system libraries. Our system reduces the mistakes and errors that can be occurred when a software programmer directly connects software to hardware, and supports balancing labors between hardware developers and software programmers. Moreover, this system can be used as the back-end for a hardware/software co-design framework.

Keywords: *Embedded System, Hardware Controller, Device Driver, Code Generation, Co-design*

1. Introduction

Every embedded system is a compact composition of hardware and software. A large portion of the embedded system development process involves the integration of hardware and software. Unfortunately, communication across the hardware/software boundary is tedious and error-prone to create. During the embedded system development process, most of the errors arise from the integration of hardware and software [1].

We propose an automatic hardware/software interface generation system for embedded systems. Embedded system designers define the specification files which are the description of the hardware functions. Each hardware component has three XML specification files – *API, Driver, and Controller*.

The function of the hardware can be described with automata. We introduce *Hardware Interface Automata* which is extended from the Labeled Transition System [10]. Hardware Interface Automata describes hardware usage for software programmers.

Our system generates specific hardware/software interface routines automatically: *Driver API*, *Device Driver*, and *Device Controller*. Therefore, embedded software programmers can easily use hardware just like pre-existing C libraries. Low level mechanism of communication is invisible to the software programmers. That is, embedded software programmers do not need to know the low level characteristics of hardware such as memory mapped I/O or kernel functions.

Automatic hardware/software interface generation reduces the mistakes and errors that can be occurred when software programmers must directly connect software to hardware. Consequently our system makes the embedded system development process faster and safer. In addition, this system can be used as the back-end for hardware/ software co-design framework. The coverage of this system is a Linux machine with Intel XScale CPU, and the additional hardware is attached in the Variable Latency I/O control area.

This paper is organized as follows. Section 2 describes other hardware interface generation systems. Section 3 describes our interface modeling, how to derive the *Interface Specification*, and code generations. Section 4 shows the implementation. The case study is given in Section 5. The conclusion and future works are presented in Section 6.

2. Related Work

Unlike other hardware/software co-design systems which focused on system specification, verification, simulation, and mapping on the target architecture, our system focuses on interface generation from the hardware/software co-design methodologies.

COSMOS [2], COSYMA [3], and CoWare [4] use concurrently-running processes using remote procedure calls (RPC) for communications. This communicating mechanism is more complicated than shared memory and raises efficiency issues.

Manuscript received September 30, 2005; accepted November 23, 2006.
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)
Corresponding Author: Choonho Son
* Network Technology Laboratory, Korea Telecom (choonho@kt.co.kr)
** Dept. of Computer Science, Korea Advanced Institute of Science and Technology (dolgam@pllab.kaist.ac.kr, hgkang@ropas.kaist.ac.kr, han@cs.kaist.ac.kr)

Interface generation in PISH [5] introduced a layered interface model like service, message, driver, and register transfer level. Despite the well-defined layered approach, this system is not a fully automatic interface generator: the designer has to manually build all libraries of low-level interface elements.

Approaches to automatic interface generation include the CHINOOK [6] approach, as well as domain-specific approaches like in the POLIS environment [7]. As in POLIS, control-dominated systems are described by a set of co-design finite state machines (CFSMs).

SHIM [8] has the most similar features with our system. SHIM uses bus-based communication with shared memory. Communication by shared memory is much simpler than by remote procedure call (RPC) or FIFO-based one. The main purpose of SHIM is integrating software and hardware using single specification language. Interface generation is the second part. The communication media - shared memory - does not have any information about hardware. It is just a communication channel between software and hardware in SHIM. Our system has the abstract information about hardware which indicates what functions it has.

3. Hardware/Software Interface

3.1 Co-design Framework

Most co-design projects have their own high-level description language. For example, POLIS [7] accepts Esterel [9] language which is translated into CFSM for each specified module or POLIS Software Hardware Interface Format (SHIFT) for hardware/software interface modules. This makes it hard to introduce pre-existing modules which are implemented by C/C++ or Verilog language.

Rather than proposing a new syntax for the hardware/software interface description such as SHIFT, we choose the simple DTD-less XML format which only includes method names, input/output signals, and additional information because the purpose of our system is to hide the low level mechanism of hardware/software communication and to offer the C programming style to the programmer. The grammar of the *Interface Specification* is accessible at our web page [12].

Fig. 1 shows the hardware/software co-design framework of our system. An embedded system designer or front-end of the co-design framework has only to make three *Interface Specification – API, Driver, and Controller* - files from the high-level hardware specifications or hardware manual. The *API specification* is equivalent to the *Hardware Interface Automata* which describes how to use hardware. The *Driver specification* describes basic communication methods between software and hardware. The *Driver API* uses *Device Driver* to communicate with the hardware. The *Controller* specification describes how many inputs or outputs exist in the hardware. The *Architecture Information* has many clues about the overall system such as the type of CPU, operating system, and system bus architecture.

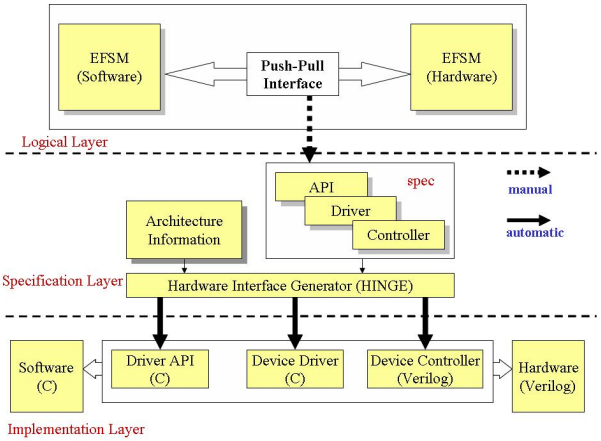

Fig. 1. Co-design Framework

Our system gathers this information and generates the *Driver API, Device Driver,* and *Device Controller* code.

3.2 Architecture of an embedded system

The embedded system is quite ambiguous to define. We assume the embedded system has an operating system and application specific hardware which is directly connected to the system bus. Fig. 2 is our model of an embedded system with FPGA.

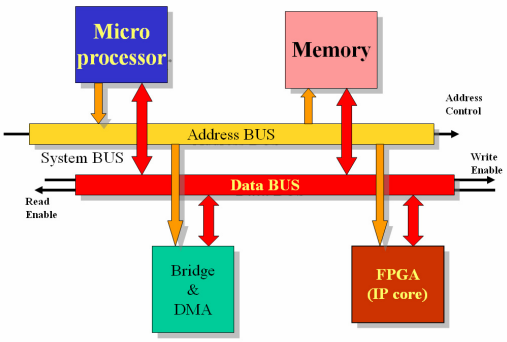

Fig. 2. Model of embedded system with FPGA

The hardware is embedded in the FPGA which has additional controller logic, including address decoder and data decoder.

There are two big modules for code generation – software generation and hardware generation. The difference between them is that hardware generation is not related with software types such as the operating system. However, software generation is tightly related with architecture information, for example, the type of CPU and operating system.

The software part of the interface has the following structure. The device driver logic is dependent on the operating system. We currently support Linux (Fig. 3).

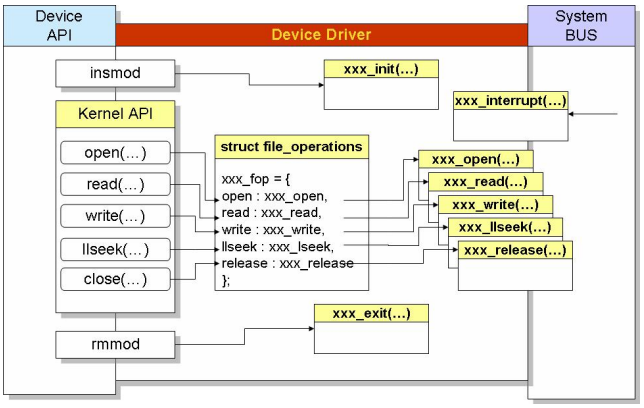

Fig. 3. Software structure of the interface

To specify the software part of the interface, we have two rules. In Fig. 4, device API indicates how to call the kernel function. In Fig. 5, DRIVER decides how to transfer the data between API and kernel.

$$
\begin{aligned}
API &::= DEVICE\ \ AUTOMATA \\
DEVICE &::= FUNC^{+} \\
FUNC &::= NAME\ PARAM^{*} \\
&\ \ \mid\ NAME\ PARAM^{*}\ RETURN \\
PARAM &::= TYPE\ NAME\ SIZE\ OP \\
RETURN &::= TYPE\ NAME\ SIZE\ OP \\
NAME &::= Identifier \\
TYPE &::= bool \mid uint8 \mid uint16 \mid char \mid char* \mid uint8* \mid uint16* \\
SIZE &::= PositiveInteger \\
OP &::= push \mid pull \\
AUTOMATA &::= STATE\_NUMBER^{+} \\
STATE\_NUMBER &::= CURRENT\_STATE\ \ RELATION^{+} \\
RELATION &::= NAME\ \ ACTION^{+} \\
ACTION &::= COND\ \ NEXT\_STATE \\
COND &::= EXPR \mid always \\
EXPR &::= NAME\ OP\ Number \\
OP &::= < \mid > \mid <= \mid >= \mid != \mid == \\
CURRENT\_STATE &::= PositiveInteger \\
NEXT\_STATE &::= PositiveInteger
\end{aligned}
$$

Fig. 4. API specification grammar

$$
\begin{aligned}
DRIVER &::= EVENT\ \ EVENT\_LAYOUT \\
EVENT &::= NAME\ \ SW2HW^{*}\ HW2SW^{*}\ CTL \\
&\ \ \mid\ NAME\ \ SW2HW^{*}\ HW2SW^{*} \\
SW2HW &::= SW\_VARIABLE\ \ HW\_PORT \\
HW2SW &::= SW\_VARIABLE\ \ HW\_PORT \\
CTL &::= interrupt \mid polling\ PIN\_NUMBER\ HW\_PORT \\
SW\_VARIABLE &::= TYPE\ NAME \\
NAME &::= Identifier \\
TYPE &::= bool \mid uint8 \mid uint16 \mid char \mid char* \mid uint8* \mid uint16* \\
HW\_PORT &::= NAME \mid NAME\ WIDTH \\
WIDTH &::= [PositiveInteger : PositiveInteger] \\
PIN\_NUMBER &::= PositiveInteger \\
EVENT\_LAYOUT &::= NAME\ VirtualAddress\ BUS\_SIZE\ OFFSET\_COUNT\ OFFSET^{*} \\
BUS\_SIZE &::= 16 \mid 32 \\
OFFSET\_COUNT &::= PositiveInteger \\
OFFSET &::= COUNT\ NAME\ BITMAP\ to\ HW\_NAME \\
&\ \ \mid\ COUNT\ NAME\ BITMAP\ from\ HW\_NAME
\end{aligned}
$$

Fig. 5. Driver specification grammar

The *Device Driver* is highly target-machine dependent. We generate the module-based device driver in the Linux kernel 2.4. We support char devices, and blocking or non-blocking I/O's. There is a well-defined API for the device driver programming. The core APIs are *open, close, read, write* and *seek*. The Driver API is a collection of functions which constitute a set of core APIs.

The hardware embedded in the FPGA is divided into three types from the viewpoint of software: 1) *write module*, 2) *read module*, and 3) *read/write module*

The *write module* and *read module* have uni-directional communication channels between software and hardware. The software writes data into the hardware, and the hardware does not send any data into the software in the *write module*. These read and write modules of the embedded system are depicted in the controller block diagrams of Fig. 6 and Fig. 7.

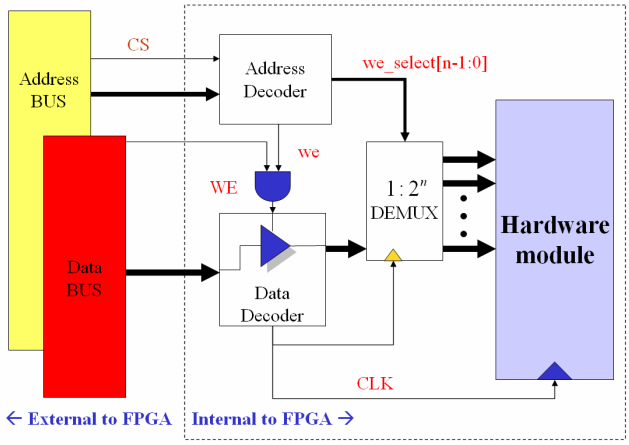

Fig. 6. Block diagram of *write model*

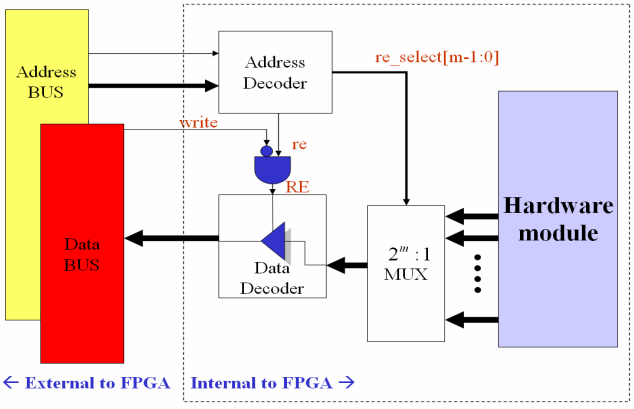

Fig. 7. Block diagram of *read model*

The *read/write module*, which has a bidirectional communication channel, is a union of *read module* and *write module*. The arbitration for reading or writing is decided by *Address Decoder* (Fig. 8).

Our system generates *Address Decoder*, *Data Decoder, Demux*, *Mux,* and *top module* in synthesizable Verilog code. The *top module* is the union of *Address Decoder, Data Decoder, Demux*, and *Mux,* as in the block diagram of Fig.

8. The Controller specification concerns how many inputs and outputs for the hardware module exist. If there are only inputs, the generated code is *write module*, which does not have *Mux* logic.

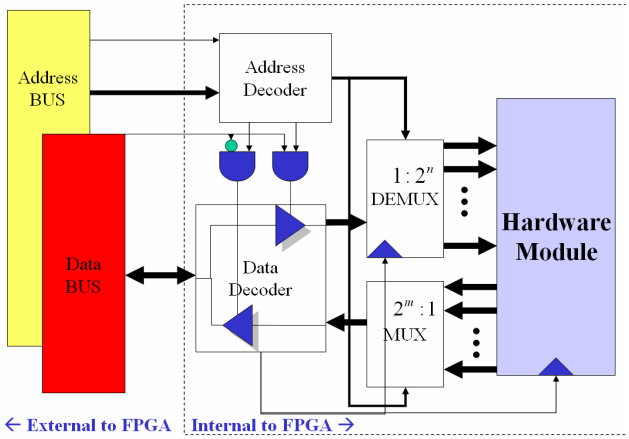

Fig. 8. Block diagram of *read/write model*

Our interface generation system has a read/write-generator module for one byte and two bytes. This size of data decoder depends on the system bus size. If the data size for reading or writing is larger than the system bus size, the software generator generates several read/write methods for the appropriate addresses. All data can be represented by one/two-byte read or write methods.

The reading methods are more complicate than the write method, because we have a selection problem for the data validity of hardware: the first concerns reading data without checking data validity; the other concerns reading data when software convinces itself of data validity. This can be implemented by polling or interrupt. We support all three methods. For simplicity, the interrupt line is directly connected with CPU, which means the interrupt line does not share with other hardware and does not use Programmable Interrupt Controller (PIC).

4. Implementation

The reconfigurable hardware (FPGA) is attached to the system bus. Intel designed the XScale micro processor for embedded systems, and the customized hardware can be connected on a variable latency I/O control. It consequently makes the communication style asynchronous, memory-mapped I/O and non-buffered. Fig. 9 shows our embedded board. There are two parts, main board (HBE-EMPOS II) and expansion kit (HBE – EMP2CYC) [11].

The code generator is implemented by the Python language. The specification is DTDless XML, which is parsed by the python DOM library. The source codes are divided into five parts. The total size of our system is depicted in Fig. 10.

Fig. 9. Embedded board and expansion kit

	API	Driver	Controller	make file	Main frame work	Total
# of line(Python)	363	1169	988	44	81	2645

Fig. 10. Size of code generator

5. Case Study

We design a car controller as an example of the hardware/software co-design. A car dashboard consists of speedometer, counter, oil-gauge and some other signals. This controller has an additional safety feature that monitors the status of the door (whether it is closed or open) and sends out a warning signal when the door is open while the car is in motion. Moreover, while the car starts, the controller ensures that the doors are closed. The state diagram of the controller is illustrated in Fig. 11.

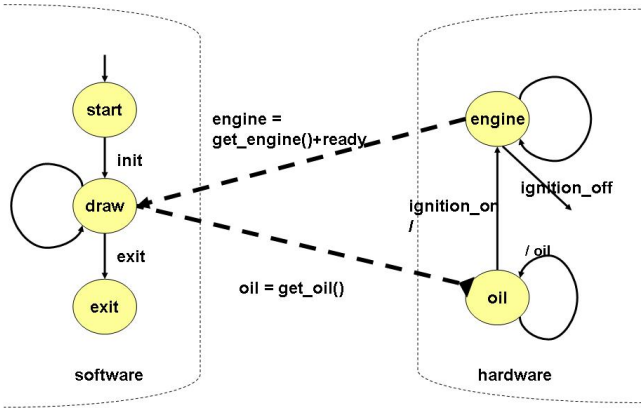

Fig. 11. Logical concept: car controller

From Fig. 11, the designer can make the specification file in XML. The size of the specifications is summarized in Fig. 12.

	API	Driver	Controller	Total
# of line(engine)	14	11	10	35
# of line(oil)	14	10	9	33
Total # of lines	28	21	19	68

Fig. 12. The number of lines: XML

The generated file size from the car controller specification of Fig. 12 is illustrated in Fig. 13.

	API(C)	Driver(C)	Controller(Verilog HDL)	Makefile	Total
# of line(engine)	36	123	75	15	249
# of line(oil)	36	95	74	15	220
Total # of lines	72	218	149	30	469

Fig. 13. Size of generated codes

The real prototype which is embedded in our board is shown in Fig. 14. The left LCD window shows car controller GUI. The right part is the expansion KIT (HBE – EMP2CYC). Pushing the dip switch, the speedometer is immediately updated, because the software (GUI) and the hardware (dip switch) are connected by our interface generated codes. The Verilog code is compiled by *Quartus II*.

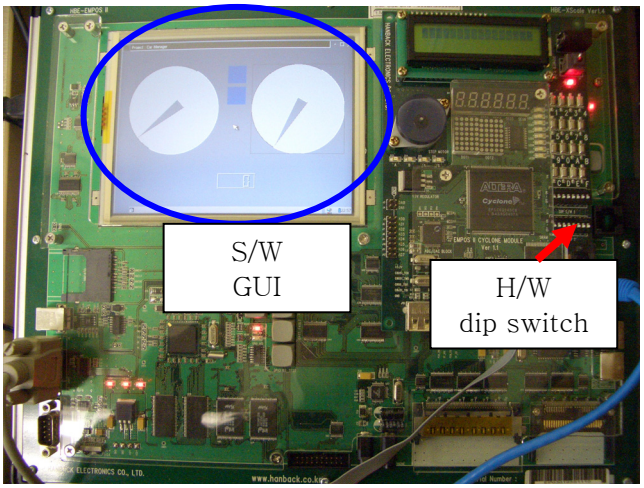

Fig. 14. Prototype of car controller

6. Conclusion and future work

This paper presents the automatic interface generation system using the *Interface Specification*. Embedded software designers can use hardware resources through customized library calls in the same way as C libraries. Many hardware/software co-design frameworks propose hardware dependent interface models or only provide low-level communications such as send or receive. In our paper, we propose a general model of the embedded system which has FPGA for hardware.

There are a few things which can be automated. One of the tedious aspects of hardware synthesis is assigning every hardware input/output port a specific hardware pin number. According to each EDA tool, including Quartus, the method for specifying pin numbers is slightly different. We can automate this method by specifying the names of the EDA tools. We do not currently manage the entire physical memory space, so the designer has to specify the physical or virtual address of the specific hardware. This work can be automated, if the hardware/software co-design framework manages the entire resources of the embedded system.

In this paper, we do not mention optimization, because our purpose is to generate an automatic interface for rapid prototyping. If our system is used for the final product, we have to optimize the memory layout of the communication events and the soundness of the generated codes.

We think that one interface generator cannot support all of the embedded system architectures because of the extraordinary characteristics of all the architectures. If the embedded board vendors service their own interface generators, then embedded system development will become dramatically easier.

Our ultimate goal is an embedded system development environment for developers. Future works are as follows. A sequence of *Hardware Interface Automata* from the start state is the right method of using hardware. We do not currently check for the illegal usage of hardware. We believe that the *Driver API* can detect the illegal path by keeping the state of the *Hardware Interface Automata* on run-time; the return value of *Driver API* can be used to check for the misuse of hardware functions.

References

[1] Choonho Son, Jeong-Han Yun, Hyun-Goo Kang, and Taisook Han, "Hardware/Software Interface Generation for Embedded System using Hardware Interface Automata," *Proceedings of the 4th International Conference on Asian Language Processing and Information Technology,* Bangkok, Thailand, 2005.

[2] Ismail, T. B., Abid and A. Jerraya, "COSMOS: A codesign approach for communicating systems," *Proceedings of the 3rd International Workshop on Hardware/software Co-Design*, Grenoble, France, 1994, pp. 17-24.

[3] Ernst, R., Henkel, T. Benner, W. Ye, U. Holtmann, D. Herrmann and M. Trawny, "The COSYMA environment for hardware/software cosynthesis of small embedded systems," *Microprocessors and Microsystems 20*, 1996, pp. 159-166.

[4] Bolsens, I., H. J. De Man, B. Lin, K. Van Rompaey, S. Vercauteren and D. Verkest, "Hardware/software co-design of digital telecommunication systems," *Proceedings of the IEEE 85*, 1997, pp. 391-418.

[5] Cristiano C. de Araujo, and Edna Barros, "Interface Generation for Concurrent Processes during Hardware/Software Co-synthesis," *Proceedings of the 15th Symposium on Integrated Circuits and Systems Design*, 2002.

[6] P. Chou, R. Ortega, and G. Borriello, "Interface co-synthesis techniques for embedded systems," *Proceedings of the IEEE/ACM International Conference on CAD (ICCAD)*, 1995, pp 280-287.

[7] F. Balarin, A. Jurecska, and H. Hsieh et al,

"*Hardware-Software Co-Design of Embedded System: the Polis Approach",* Kluwer Academic Press, Boston, 1997.

[8] Stephen A. Edwards, "SHIM: A language for Hardware/Software Integration," *Synchronous Languages, Applications, and Programming* (SLAP), 2005.

[9] Berry, G. and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming* 19, pp. 87-152.

[10] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jah, and Helmut Veith, "Modular Verification of Software Components in C," *Transactions on Software Engineering (TSE)*, Vol. 30(6), pp 388~402, June 2004.

[11] Hanback electronics homepage, http://www.hanback.co.kr

[12] Interface Description Language grammar, http://pllab.kaist.ac.kr/~chson/HIL/grammar.html

Choonho Son
He received his BS degree in Computer Science from Yonsei Univ. in 2003 and an MS degree in Computer Science from KAIST in 2005. He is now working with the Technology Laboratory, Korea Telecom. His research interests include embedded systems, co-design, and programming language.

Jeong-Han Yun
He received BS and MS degrees in Computer Science from KAIST in 2001 and 2003, respectively. He is currently undertaking a doctorate course as a member of the programming language lab at KAIST. His research interests include embedded systems, co-design, programming language, and program analysis.

Hyun-Goo Kang
He received BS and MS degrees in Computer Science from Hanyang Univ. in 1997 and 1999, respectively. During 1999~2000, he stayed with ETRI. He is now undertaking a doctorate course as a member of the programming language lab at KAIST. His research interests include embedded systems and program analysis.

Taisook Han
He received a Ph.D. degree in Computer Science from the Univ. of North Carolina at Chapel Hill in 1990. He has been a professor at KAIST since 1991. His research interests are in the areas of programming language, compilers, embedded systems, and program analysis.