Generating a Device Driver with a Formal Specification Language

Tetsuro Katayama, Keizo Saisho, and Akira Fukuda

Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5 Takayama Ikoma, Nara 630-0101, JAPAN.
{kat,sai,fukuda}@is.aist-nara.ac.jp

Abstract

This research investigates the possibility of automatic generating OS(Operating System). Writing device drivers is one of difficult tasks to develop or port OS. In this paper, the burden in writing device drivers is lighten and their productivity is improved by using a formal specification language. An automatic device driver generation system is proposed, and its inputs are discussed. The inputs for the system are determined by both of a specification and a data sheet of the device. A specification of the device shows fundamental functions of the device, and a data sheet of the device shows peculiar values of the device. Parameters of both inputs are extracted from existing device drivers. A formal specification language VDM-SL(the Vienna Development Method Specification Language) is adopted in order to describe specification of devices. As an example, a specification for a printer device is written in VDM-SL. Formal specification languages can remove obscurity from a specification of software in comparison with other programming languages, and then bugs can be reduced in the specification as many as possible. By describing a specification for devices clearly, device drivers can be generated more easily. Consequently, time and labor in generating the device drivers decrease.

Keywords: Operating systems, Device drivers, Formal specification languages, VDM-SL.

1 Introduction

OS(Operating System) cannot be adapted to various hardware or application programs, which are being developed. One of the reasons is that much time and efforts in writing device drivers are spent. As internet is grown or multi-media is progressed, various devices would be developed. It is a more serious problem to spend much time and make efforts. We should urgently cope with reducing the burden.

Writing device drivers is one of difficult tasks to develop or port OS. Some of the reasons are as follows:

- Programmers of device drivers must know information about hardware such as specifications of devices and carefully describe complex parts such as timing control.
- When two devices have different chips (controllers) even if they offer the same services, the programmers must write two different device drivers for each of them.
- If we change an OS but use the same devices, we need to rewrite the device drivers for a new OS.

Most of researches into OS concentrate on its design and/or improvement of its performance such as scheduling policy, memory management and file system construction. It is difficult to generate, modify, port, test and debug OS because its program is larger and more complex than application programs. Few studies on generating device drivers or operating systems themselves have been reported[1]-[4]. It is necessary to improve productivity of OS. Our study has investigated the possibility of automatically generating OS[5].

In this paper, we propose an automatic device driver generation system, and its inputs are discussed. In section 2, we describe an automatic device driver generation system. As an example, a specification for a printer device is written in VDM-SL(the Vienna Development Method Specification Language), which is one of formal specification languages. Formal specification languages can remove obscurity from a specification of software in comparison with other programming languages, and then bugs can be reduced in the specification as many as possible. By using a formal specification language, we aim at lightening the burden in writing device drivers and improving their productivity. In section 3, we discuss and evaluate the system. In section 4, we present our conclusion.

2 Device Driver Generation System

In this section, we propose an automatic device driver generation system.

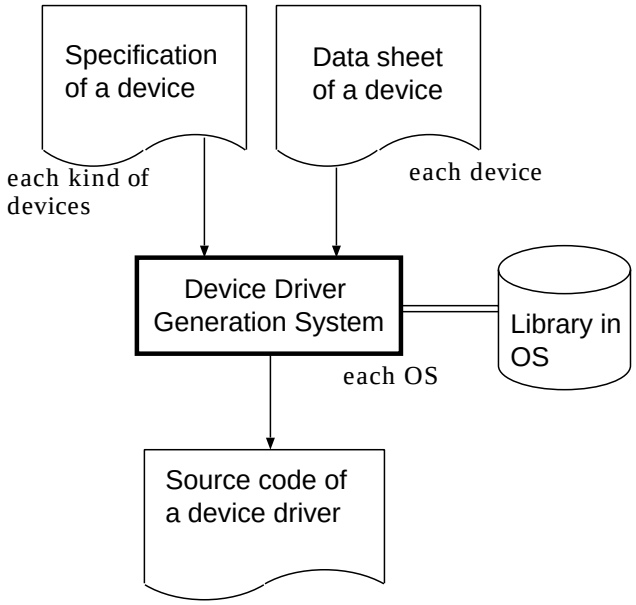

Figure 1: Outline of the device driver generation system.

2.1 Outline of the System

In developing the device driver generation system, we should abstract device driver generation itself. We consider that inputs for the system are various elements such as functions, values, timing control, and so on. We attempt to simplify inputs for the system. We divide a specification and a data sheet from a device driver whether or not it depends on a chip. The inputs for the system are determined as follows:

- A specification of a device — It shows fundamental functions of the device. It is associated with a kind of devices such as floppy disk, CD-ROM, and so on.
- A data sheet of a device — It shows peculiar values of the device such as I/O ports, control sequences, and so on. It must be prepared for each device.

Thus, when we change a device to a new one which offers the same services, we can use the same specification to give only the data sheet depending on the chip of the new one.

Figure 1 shows an outline of the system. The system generates a source code of a device driver by giving both of a specification and a data sheet of the device which we would like to generate. The system would have libraries depended on the target OS as data in the system. The libraries give how to use functions in the target OS because the way of calling device drivers differs in each OS.

As an example of the target device in this paper, we choose a printer device. It is simpler and its specification which we need to write is smaller than other devices. We examine existing printer device drivers in MINIX[6], FreeBSD[7] and NetBSD[8] and investigate the abstraction of the drivers to extract peculiar values and fundamental functions of the driver from them. Table 1 shows the functions for printer devices.

Table 1: Functions for printer devices

Name	Action
init	initializing devices
open	making devices possible
close	making devices impossible
write	writing values on devices
cancel	canceling the writing operation
error	handling errors

As similar, Table 2 shows peculiar values which we extract for printer devices.

2.2 Specification in VDM-SL

We adopt a formal specification language in order to describe the specification of the device which is wanted to generate. It can remove obscurity from the specification in comparison with other programming languages, and then bugs can be reduced in the specification as many as possible.

We choose VDM-SL (the Vienna Development Method Specification Language)[9] as the language to describe a specification of devices. A specification written in VDM-SL can be translated into a source code written in programming language C++. It can prevent bugs from getting into the code and we can easily execute it.

Considering Table 1 and Table 2, we write a sample specification of a printer device written in VDM-SL as Figure 2 and 3. In Figure 2, states are expressed by using global variables and the state transit is presented by using operations. Hence, any operations can use global variables by assignment of values such as port, status and command to them. Because interrupts and timing control cannot be described in VDM-SL, we cannot generate such parts automatically. In this paper, we generate a rough process in a device driver for the target device. In Figure 2, a hexadecimal is expressed by using the prefix "0x" to be easily understandable because VDM-SL cannot treat with a hexadecimal type. Also, the specification given by Figure 3 includes ambiguous parts in post conditions as sentence structure of VDM-SL, but we do not describe in detail.

Figure 4 shows a part (Init part) of a source code written in C++, which is translated the sample speci-

Table 2: peculiar values of printer devices

Name	Part
data register port	the port to give print data
status register port	the port where status of a printer are read or written
control register port	the port to give control codes
control code	a command to control the printer
status code	a value which denote status of the printer
interruption request (IRQ)	the number of interrupts

```
values
  Data_Port    = 0x3BC;
  Status_Port  = 0x3BD;
  Control_Port = 0x3BE;

  assert_strobe_Command = 0x1D;
  negate_strobe_Command = 0x1C;
  select_Command        = 0x0C;
  init_printer_Command  = 0x08;

  busy_Status    = 0x10;
  nopaper_Status = 0x20;
  normal_Status  = 0x90;
  online_Status  = 0x10;

  printer_IRQ = 7
```

Figure 2: Sample values of a printer device written in VDM-SL

fication in VDM-SL. The approximate execution time in the translation measures 0.031 sec. on SUN Ultra 1.

3 Discussion

In this section, we discuss the specification language, which we adopt as a language to describe inputs for the system, the efficiency of the system, the standard interface I_2O(Intelligent Input Output), and devices written and designed with HDL(Hardware Description Language).

3.1 Description by Formal Specification Languages

A formal specification language VDM-SL is adopted as a language to describe inputs for the device driver generation system. It can remove obscurity from the specification in comparison with other programming languages, and then bugs can be reduced in the specification as many as possible.

Generally, formal specification languages has less powerful to describe than programming languages. It is convenient that VDM-SL can treat with various data types easily. It, however, executes a process by calling functions or procedures. The order in process execution must be described in the order of the called functions or procedures.

LOTOS (Language Of Temporal Ordering Specification) [10] is popular one of formal specification languages. Figure 5 and 6 shows sample values and specification of printer device written in LOTOS. Because in LOTOS, abstract data types can be only defined, constants must be generated by operations and it is complicated (as Figure 5). Also, it is difficult to make variables corresponded to state variables in programming languages because passes by values are explained through interaction among processes.

In this paper, we adopt VDM-SL as a formal specification language. If there was the language which get both functions of declaration of constants such as port number and description of the order of a process, it might be more easy to describe a specification of devices.

3.2 Efficiency of the System

By describing the specification clearly, the printer device driver can be generated more easily. Moreover, the specification of devices is not needed to rewrite when we adapt a new device of which functions are equal to those of the old one. Consequently, the burden which we need in writing device drivers can decrease, and their productivity is improved.

We cannot, however, generate a complete device driver automatically. At the end of generation,

(A) we need to write manually the interrupts and timing control, which cannot are described in the specification in VDM-SL, in the generated source code, and

(B) we need to check both of the number and names of arguments in each function or procedure.

About (A), we consider how both parts of interrupts and timing control are generated. In order to automatically generate interrupts, we plan to use and apply the libraries in the system. As to timing control, it is realized by various ways in existing devices. If timing control of a device is realized by hardware, we need not write specifically it on a device driver of the

```
operations
  Print_drv() ==
  (
    Init();
    while true do
      ||(Open(),Close(),Write(),
          Cancel(),Error())
  );

  Init() ==
  (
    out_byte(Control_Port,
             init_printer_Command);
    out_byte(Control_Port,
             select_Command);
    put_irq_handler(printer_IRQ,
                    Print_handler)
  )
  ext rw status
  pre status = online_Status;

  Open() ==
  ext rw status
  post status = online_Status;

  Close() ==
  ext rw status
  post status = not online_Status;

  Write() ==
  (
    if (data_count = 0) then
      out_byte(Control_Port,
               select_Command)

    out_byte(Data_Port, data);
    out_byte(Control_Port,
             assert_strobe_Command);
    out_byte(Control_Port,
             negate_strobe_Command);
  )
  ext rw status;
      ro data_count
  pre data_count >= 0 and
      status = normal_Status;

  Cancel() ==
  ext rw data_count
  post data_count = 0;

  Error() ==
    case status :
      busy_Status -> retry(),
      not online_Status and nopaper_Status
        -> paper_empty(),
      not online_Status
        -> out_byte(Control_Port,
                    select_Command)
```

Figure 3: A sample specification of a printer device written in VDM-SL

device. Thus, we can use the system without modification. By contrast, if timing control should be realized by software, we write it on a data sheet, which is one of the inputs of the system. We plan to generate automatically a device driver by using timing control on the data sheet as it is. According to circumstances, we may develop a more adaptable language for writing device drivers.

About (B), we had to make adjustments of the number or names of arguments in each function or procedure. We consider that we can prevent from adjusting by writing a specification more minutely, which is one of the inputs of the system.

```
void vdm_DefaultMod_Init() {
  PushFile("printer.vdm");
  {
    PushPosInfo(34, 3);
    {
      PushPosInfo(35, 13);
      Int tmpVar_14;
      tmpVar_14 =
        vdm_DefaultMod_Control__Port;
      Int tmpVar_15;
      tmpVar_15 =
        vdm_DefaultMod_init__printer__Command;
      vdm_DefaultMod_out__byte
        (tmpVar_14, tmpVar_15);
      PopPosInfo();
    }
    {
      PushPosInfo(37, 13);
      Int tmpVar_16;
      tmpVar_16 =
        vdm_DefaultMod_Control__Port;
      Int tmpVar_17;
      tmpVar_17 =
        vdm_DefaultMod_select__Command;
      vdm_DefaultMod_out__byte
        (tmpVar_16, tmpVar_17);
      PopPosInfo();
    }
    {
      PushPosInfo(39, 20);
      Int tmpVar_18;
      tmpVar_18 =
        vdm_DefaultMod_IRQ;
      Int tmpVar_19;
      tmpVar_19 =
        vdm_DefaultMod_print__handler;
      vdm_DefaultMod_put__irq__handler
        (tmpVar_18, tmpVar_19);
      PopPosInfo();
    }
    PopPosInfo();
  }
  PopFile();
}

Bool vdm_DefaultMod_pre__Init() {
  PushFile("printer.vdm");
  PopFile();
  return (Bool)
    (vdm_DefaultMod_status ==
     vdm_DefaultMod_online__Status);
}
```

Figure 4: An 'Init part' of a translated source code in C++ from the specification in VDM-SL

3.3 Standard Interface I_2O

I_2O (Intelligent Input Output) SIG[11] has determined standard interface I_2O between OS and devices[12]. Under the specification of I_2O, a device driver is divided into three sections as follows: OSM (OS Specific Module), HDM (Hardware Device Module) and Messenger. OSM and HDM are depended on OS and hardwares (devices), respectively. Messenger communicates between them. I_2O specifies a form of packets used at

```
specification print_drv
  [init,select,put_handler,
   data,strobe,nstrobe,
   retry,paper_empty] : noexit
type Ports is
  sort hex
  opns Data_Port    : -> hex
       Status_Port  : -> hex
       Control_Port : -> hex
  eqns ofsort hex
       Data_Port    = 0x3BC;
       Status_Port  = 0x3BD;
       Control_Port = 0x3BE;
endtype
type Command is
  sort hex
  opns assert_strobe_Command : -> hex
       negate_strobe_Command : -> hex
       select_Command        : -> hex
       init_printer_Command  : -> hex
  eqns ofsort hex
       assert_strobe_Control = 0x1D;
       negate_strobe_Control = 0x1C;
       select_Control        = 0x0C;
       init_printer_Control  = 0x08;
endtype

type Status_register is
  sort hex
  opns busy_Status    : -> hex
       nopaper_Status : -> hex
       normal_Status  : -> hex
       online_Status  : -> hex
       mask_Status    : -> hex
  eqns ofsort status
       busy_Status    = 0x10;
       nopaper_Status = 0x20;
       normal_Status  = 0x90;
       online_Status  = 0x10;
endtype

type Interrupt_Request is
  sort int
  opns irq : -> int
  eqns ofsort int
       irq = 7;
endtype
```

Figure 5: Sample values of a printer device written in LOTOS

communication between OSM and HDM. OS can communicate the device without rewriting its HDM even if OS changes.

If we adopt it to the proposed device driver generation system, we need not to prepare the system every target OS for which we develop the system. It becomes easily to develop the system itself. It, however, generates overheads by dividing a device driver into classes and includes lower performance than usual. The overheads becomes critical, in particular, in devices moved at high speed. Hence, we must consider whether or not we use it in our proposed system.

3.4 Devices Designed with HDL

In order to generate device drivers by our proposed system, we need to give a specification and a data sheet of devices to the system. In respect of devices written and

b

```
behaviour
     print_init[init,select,put_handler]
  >> (write[data,strobe,nstrobe,
            select,retry,paper_empty]
   [] error[data,strobe,nstrobe,
            select,retry,paper_empty])
where
  process print_init
    [init,select,put_handler]: exit :=
      init;
      select;
      put_handler;
      exit
  endproc
  process write[data,strobe,nstrobe,
                select,retry,paper_empty]:
    noexit :=
      data;
      strobe;
      nstrobe;
      (write[data,strobe,nstrobe,
             select,retry,paper_empty]
    [] error[data,strobe,nstrobe,
             select,retry,paper_empty])
  endproc
  process error[data,strobe,nstrobe,
                select,retry,paper_empty]
    noexit :=
        ([busy]
          -> retry
      [] [not(online) and nopaper]
          -> paper_empty
      [] [not(online)]
          -> select);
        (write[data,strobe,nstrobe,
               select,retry,paper_empty]
      [] error[data,strobe,nstrobe,
               select,retry,paper_empty])
  endproc
endspec
```

Figure 6: A sample specification of a printer device written in LOTOS

designed with HDL(Hardware Description Language) such as Verilog-HDL[13], it may be possible to generate inputs of the proposed system from the specification itself written in HDL. In future, we adopt such devices to the system, and construct the system to generate device drivers automatically from its description.

3.5 Related Work

We research the possibility of generating OS automatically. In this paper, we propose an automatic device driver generation system, and its inputs are discussed. Few studies on generation of OS itself or parts in OS have been reported.

Jia and Maekawa claim that it is significant to construct kernel of OS automatically and that the construction method urgently is needed[1]. They, however, do not show any approach or design to realize the method.

Chou *et al.* have proposed the hardware/software co-synthesis system to automatically synthesize device driver routines as a step in the design of embedded

controllers[2]. They have exploited algorithms from graph theory to partition and schedule interface events in a viewpoint of hardware/software co-design.

4 Conclusion

In this paper, we aim at lightening the burden in writing device drivers and improving their productivity by using a formal specification language. We have proposed an automatic device driver generation system. We divide a specification and a data sheet from a device driver whether or not it depends on a chip. The system generates a source code of a device driver by giving both of a specification and a data sheet of a device. A printer device is chosen as an example. We examine existing printer device drivers and investigate the abstraction of the drivers to extract peculiar values and fundamental functions of the driver from them. In addition, we write the specification of a printer device driver in VDM-SL, which is one of formal specification languages. By using a formal specification language, we can remove obscurity from a specification of software in comparison with other programming languages, and then bugs can be reduced in the specification as many as possible. As a result,

- a rough process in a device driver for the target device can be generated automatically,
- the specification of devices, which is input of the system, is not needed to rewrite when we adapt a new device of which functions are equal to those of the old one, and
- because of the above, the burden which we need in writing device drivers can decrease, and their productivity is improved.

Future issues are as follows:

- Enhancement of the system.

  We cannot generate a complete device driver automatically because we do not treat with interrupts or timing control in this paper. We must enhance the system to be able to generate automatically device drivers completely. According to circumstances, we may develop a more adaptable language for writing device drivers.

- Improvement of adaptability to the system.

  In this paper, a printer device is chosen as an example because it is smaller and simpler than other devices. We need to adapt them to the system referring to existing device drivers in order to show usefulness of the system. We plan to adapt network adapters of which many kinds exist.

- Adoption of standard interface I_2O (Intelligent Input Output)[11][12].

  I_2O SIG has determined standard interface I_2O between OS and devices. Under the specification of I_2O, OS can communicate the device with the same device driver even if OS changes. It, however, includes lower performance than usual. We must consider whether or not we use it in our proposed system.

- Generation of device drivers for devices written in HDL (Hardware Description Language).

  We need to give a specification and a data sheet of devices to the system. In future, we adopt devices written in HDL such as Verilog-HDL[13], and construct the system to generate device drivers automatically from its description.

References

[1] X. Jia and M. Maekawa: "Operating System Kernel Automatic Construction," *Operating Systems Review*, Vol.29, No.3, pp.91-96, 1995.

[2] P.H. Chou, R.B. Ortega, G. Borriello: "The Chinook Hardware/Software Co-synthesis System," *Proc. 8th Int'l Symp. on System Synthesis*, pp.22-27, 1995.

[3] S. Ritz, M. Pankert, J. Waltenberger, V. Zivojnovik, and H. Meyr: "Code Generation Techniques in the Block Diagram Oriented Design Tool COSSAP/DESCARTES, " *Proc. 5th Int'l Conf. on Signal Processing Applications and Technology,* Vol.1, pp.709-714, 1994.

[4] E. Tuggle: "Introduction to Device Driver Design," *Proc. 5th Annual Embedded Sys. Conf.*, Vol.2, pp.455-468, 1993.

[5] T. Katayama, K. Saisho and A. Fukuda: "A Method for Automatic Generation of Device Drivers with a Formal Specification Language," *Proc. Int'l Works. on Principles of Softw. Evolution (IWPSE98)*, pp.183-187, 1998.

[6] A.S. Tanenbaum: "Operating System - Design and Implementation," *Prentice Hall*, 1987.

[7] FreeBSD Inc.: http://www.freebsd.org/

[8] NetBSD Project: http://www.netbsd.org/

[9] C.B. Jones: "Systematic Software Development using VDM," *Prentice Hall*, 1990.

[10] T. Bolognesi and E. Brinksma: "Introduction to the ISO Specification Language LOTOS," *Comp. Netw. ISDN Sys.*, Vol.14, No.1, pp.25-59, 1987.

[11] I_2O SIG: http://www.i2osig.org/

[12] D. Wilner: "I_2O's OS Evolves," *BYTE, Int. Ed., McGraw-Hill*, Vol.23, No.4, pp.47-48, 1998.

[13] D.E. Thomas and P.R. Moorby: "The Verilog Hardware Description Language (2^{nd} ed.)," *Kluwer Academic Publishers*, 1995.