A Method for Automatic Generation of Device Drivers with a Formal Specification Language

Tetsuro Katayama\textsuperscript{1}, Keizo Saisho and Akira Fukuda
\{kat, sai, fukuda\}@is.aist-nara.ac.jp

Graduate School of Information Science,
Nara Institute of Science and Technology
8916-5 Takayama Ikoma, Nara 630-0101, JAPAN.

1 Introduction

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

Writing device drivers is the most difficult one among processes to develop or port OS. The reasons are as follows:

- Programmers must know information about hardware such as specifications of devices and have carefulness to code complex parts such as timing control.

- When two devices have different chips (controllers) even if they offer same services, programmers must write two different device drivers adapted to each of them.

- If we change OS but use the same devices, we need to rewrite the device drivers adapted to new OS.

Most of researches into OS, however, concentrate on its design and/or improvement of its performance such as scheduling policy, memory management and file system construction. Few studies on generation of OS itself or parts in OS have been reported\cite{1}\cite{2}. It is necessary to improve productivity of OS. Our study has investigated the possibility of generating OS automatically.

In this paper, we aim at lightening the burden in writing device drivers and improving their productivity. We propose an automatic device driver generation system, and its inputs are examined.

2 Device Driver Generation 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. If we separate a part dependent on a chip from a device driver, we need not change fundamental functions of the device which are not depended on the chip. And then, when we change a device to a new one which offers the same services, we have

\textsuperscript{1}Tel. +81-743-72-5323, Fax. +81-743-72-5329
to rewrite only the part depended on a chip of the new one. As a result, we can lighten the burden in developing device drivers. Hence, we determine the inputs for the system as follows:

- A specification of the device — It shows fundamental functions of the device. It is associated with kinds of devices such as floppy disk, CD-ROM, and so on.
- A data sheet of the device — It shows peculiar values of the device. It must be prepared for each device.

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 a device. The system would have libraries depended on the target OS as data in the system. Because the way of calling device drivers differs in each OS, the libraries give how to use functions in the target OS.

Figure 1: Outline of the device driver generation system.

As an example, we choose a printer device because it is simpler and its specification which we need to write is smaller than other devices. Referring to existing printer device drivers, we extract fundamental functions from them. Table 1 shows the functions for printer devices.

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

We adopt a formal specification language in order to write the specification of devices. Because, we can remove obscurity from the specification by using formal specification languages 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)[3] as language to describe a specification of devices. Because in LOTOS (Language Of Temporal Ordering Specification)[4], which is popular one of formal specification languages, abstract data types can be only defined, constants must be generated by operations and it is complicated. Also, it is difficult to make variables
Table 1: Functions for printer devices

<table>
<thead>
<tr>
<th>Name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>Initializing devices</td>
</tr>
<tr>
<td>open</td>
<td>Making devices possible.</td>
</tr>
<tr>
<td>close</td>
<td>Making devices impossible.</td>
</tr>
<tr>
<td>write</td>
<td>Writing values on devices.</td>
</tr>
<tr>
<td>cancel</td>
<td>Cancelling the writing operation</td>
</tr>
<tr>
<td>error</td>
<td>Handling errors</td>
</tr>
</tbody>
</table>

Table 2: peculiar values of printer devices

<table>
<thead>
<tr>
<th>Name</th>
<th>Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>data register port</td>
<td>the port to give print-data</td>
</tr>
<tr>
<td>status register port</td>
<td>the port where status of a printer are read or written</td>
</tr>
<tr>
<td>control register port</td>
<td>the port to give control codes</td>
</tr>
<tr>
<td>control code</td>
<td>commands to control a printer</td>
</tr>
<tr>
<td>status code</td>
<td>values which denote status of a printer</td>
</tr>
<tr>
<td>interruption request (IRQ)</td>
<td>number of interrupts</td>
</tr>
</tbody>
</table>

corresponded to state variables in programming languages because passes by values are explained through interaction among processes. Hence, we do not choose LOTOS.

Furthermore, a specification written in VDM-SL can be translated into a code written in programming language C++. Hence, it can prevent bugs from getting into the code and we can execute it easily.

Considering Table 1 and Table 2, we write a sample specification of a printer device written in VDM-SL as Figure 2. 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. Strictly speaking, the specification given by Figure 2 includes ambiguous parts in sentence structure of VDM-SL, but we do not describe in detail.

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.

Because we cannot generate a complete device driver automatically, at the end of generation,

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

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

operations
Print_drv() ==
  (
    Init();
    while true do
      if (Open(), Close(), Write(), Cancel(), Error()) then
        break;
      end
    end
  )

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);
  end)

ext rw status;
re 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 2: A sample specification of printer device written in VDM-SL

About (A), we consider how we generate both parts of interrupts and timing control. In order to 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 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 a number or names of arguments in each function. We consider that we can prevent from adjusting by writing a specification more minutely, which is one of the inputs of the system.

3 Conclusion

In this paper, we aim at lightening the burden in writing device drivers. We propose an automatic device driver generation system. The system generates a source code of a device driver by giving both of a specification and a data sheet of a device. we choose VDM-SL, which is one of formal specification
languages, to describe a specification of the device. 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, we chose a printer device as an example because it is smaller and simpler than other devices. We need to adapt other devices to the system referring to existing device drivers.

- Adoption of standard interface I2O (Intelligent Input Output)[5].
  
  I2O SIG has determined standard interface I2O between OS and devices. Under the specification of I2O, OS can communicate the device with the same device driver even if OS changes. It, however, includes lower performance than usual. Hence, we must consider whether or not we use it in the system we propose.

- 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[6], and we construct the system to generate device drivers automatically from its description.

References


