システムソフトウェアと
オペレーティング・システム 73－18
（1996．8．27）

OSの自動生成に向けて

長尾 周司　片山 徹郎　張 漢明
最所 圭三　福田 晃

奈良先端科学技術大学院大学
情報科学研究科
e-mail:{syuuzi-n, kat, kanmei-c, sai, fukuda}@is.aist-nara.ac.jp

オペレーティングシステム（OS）の仕様からOS自体を自動的に生成する事により，異なるアーキテクチャに移植する際の手間を削減する事ができる．その中で，デバイスドライバはハードウェアに依存しているため，OSを移植する際に最も手間がかかる．そこで今回は，OSの自動生成としてデバイスドライバの自動生成に対象を絞る．デバイスドライバの生成システムを作成する際に，デバイスの仕様を記述するために必要となる機能を，VDM-SLとLOTOSの二つの仕様記述言語を用いた簡単な記述例を通して考察した．その結果，二つの言語はともに抽象度に差はあるが，デバイスの仕様を記述する事が可能である．

Toward Automatic Generation of Operating Systems

Shuuji Nagao　Tetsuro Katayama　Kanmei Cho
Keizo Saisho　Akira Fukuda

Graduate School of Information Science,
Nara Institiute of Science and Technology
8916-5, Takayama-cho, Ikoma-shi, Nara 630-01, Japan

We consider a method to automatically generate device drivers, because it is difficult to port device drivers to different machines. If we could generate source codes of a operating system(OS) from specifications for it, we could easily port it from an architecture to different one. In order to develop the system to generate the device drivers, we use two specification languages (VDM-SL and LOTOS) to describe device specifications, and investigate required functions which a language provides for describing the device specifications. As an example, we show a printer device specification. As a result we find out that we could describe it with both specification languages.

1 はじめに

現在，あるアーキテクチャに実装されているオペレーティングシステム（OS）を異なるアーキテクチャに移植する場合に必要となるソースコードの書き換えは，人の手によってなされている．従来，OSの研究は，スケジューリングポリシー，メモリ管理，ファイルシステムの機能，OSの設計や性能に大きな影響を与える構成方法などに集中していたためや，その困難さ故に，OS自体を生成する方法はほとんど研究されていなかった[1]．本研究はOSの自動生成の可能性を追求する．

OSの核であるカーネル部分は，ハードウェアを直接操作するため，OSを異なるアーキテクチャに移植する際には大幅な書き換えを必要とする．しかもカーネルプログラミングのうちハードウェアに依存する部分のプログラミングは，最もコーディングおよびデバッグが難しい部分である．

実際，カーネルのどの部分がハードウェアに依存しているのかを，OSの構成法の一つであるマイクロカーネル構成を例として調べる．マイクロカーネル構成はOSをシステムサーバとマイクロカーネルとの2つの部分に分けている．システムサーバにはファイルサーバやメモリマネージャがあり，アプリケーションに対するOSへのインターフェイスとなっている．システムサーバは，マイクロカーネルとメッセージを交換することによってハードウェアを利用している．この際，マイクロカーネルを通しているため，システムサーバはハードウェアには依存しない．従って，OSを移植する際にシステムサーバは書き換える必要がない．マイクロカーネルはデバイスの操作，割込み処理やコンテキストスイッチといった機能を提供している．マイクロカーネルはハードウェアに依存する部分が多いため，OSを構築，移植する際に最もコーディングに時間がかかる．マイクロカーネルのハードウェア依存部分の中でも，最も時間と労力がかかるのはデバイスドライバの部分である．同じサービスを提供するデバイスでも，使われているハードウェアが違えば異なるドライバを提供しなければならない．

システムサーバはアーキテクチャに全く依存しないため，マイクロカーネルの構築，または異なるアーキテクチャへ移植する際のコーディングの量を減らす事が，そのままOS自身の移植にかかる手間を削減することにつながる．

デバイスの仕様記述からデバイスドライバを自動生成する事で以下の利点が生じる．

- デバイスドライバを異なるアーキテクチャに移植する際の労力が削減できる．
- 新しいデバイスに対応するデバイスドライバの作成に要する時間と労力が削減できる．
- デバイスの仕様を記述する事により，コーディングの量を減少できる．

本研究ではOSの自動生成のうち，OSを様々なアーキテクチャに実装する際に，コーディングおよびデバッグが最も困難であるデバイスドライバの自動生成に対象を絞り，デバイスの仕様記述から，その仕様を満たすデバイスドライバのソースコードを生成する事を目的とする．今回は，デバイスの仕様の記述に，VDM-SL[2]とLOTOS[3]の二つの言語を用い，デバイスの仕様の簡単な記述例を示した後，デバイスの仕様記述に用いる言語に必要な機能について考察する．

他にデバイスドライバを生成する方法として，仕様記述言語を用いてデバイスドライバの仕様を記述することが考えられる．しかし，デバイスドライバの仕様を仕様記述言語で記述した場合，コーディング量はプログラミング言語でデバイスドライバを記述した場合と変わらない．また，デバイスドライバを記述するためには，割込みを扱う機能とハードウェアを直接操作する機能が，記述に使用する言語でサポートされている必要がある．割込みの記述はLOTOSではサポートされているが，VDM-SLではサポートされていない．ハードウェアを直接操作する機能は両言語ともにサポートされていない．上記の理由により，デバイスドライバの仕様を記述する方法は，プログラミング言語でデバイスドライバを記述する方法に対して利点がなかったため，本研究では採用しなかった．

2 システムの概要

我々は，デバイスドライバを生成する処理系として図1の処理系を想定している．デバイスの仕様記述をデバイスドライバ生成システムに入力する事に

よって，仕様を満たすソースコードを生成する．このシステムによって，デバイスドライバを生成するためにはデバイスの仕様記述を与えればよいため，デバイスドライバを作成する際および異なるアーキテクチャに移植する際の時間と手間の削減を実現できる．

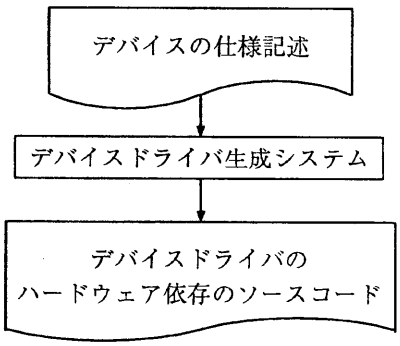

図 1: 生成処理系の概要

3 仕様記述言語による OS の記述

3.1 仕様記述言語

デバイスの仕様記述に使用する言語として今回は VDM-SL[2] と LOTOS[3] の二つを用いる．この節では，両言語について簡単に説明する．

3.1.1 VDM-SL

VDM-SL は VDM（The Vienna Development Method）の仕様記述言語部分であり，データ型から構成したモデルの状態を変化させる仕様が記述可能な，モデル指向仕様記述言語である [4]．データ型は数値（整数，自然数など），文字列などの基本型や，集合，マップなどの合成型を持ち，このデータ型により生成した値を，関数と操作で変化させる記述を行なう．

3.1.2 LOTOS

ネットワークアーキテクチャである OSI（Open System Interconnection）の標準化に伴って，仕様化を行なう必要が出て来た．これは，インプリメントの際に OSI 標準に対する曖昧な解釈が生じる事を防ぐためである．ISO で OSI 標準の仕様を記述するための形式記述技法として開発された言語の一つが LOTOS である [3][7]．イベント間の時間的な順序関係によってシステムを記述する．プロセス代数の理論に基づいて，プロセスの動作と相互作用とが記述可能である．データ構造は抽象データ型言語 ACTONE に基づいている．

次節では両言語によりデバイスの仕様の簡単な記述例を取り上げ，記述に適した言語機能を調べる．

3.2 仕様記述言語によるデバイスの仕様の記述例

今回は一例として，デバイスを記述する仕様の量が少ないプリンタデバイスについて考える．プリンタドライバを生成するために記述しなければならないデバイスの仕様として以下の項目について示す．

- コントロールレジスタポート

  コントロールコードをこのポートに出力してデバイスの初期化などを行なう．

- データレジスタポート

  印刷するデータなどをこのポートに出力する．

- ステータスレジスタポート

  プリンタの状態をこのポートを通して読み書きする．

- コマンドコード

  初期化などのコマンド．

- ステータスコード

  プリンタの状態に応じた値．

- IRQ

  Interrupt Request.

- 操作
  - init
  - write
  - error

次節では MINIX[5] のプリンタードライバを参考にした記述例を示す．

3.2.1 VDM-SL

以下はVDM-SLでの記述例である(言語の詳細については文献[6]を参照).

```
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;

  IRQ = -6

operations
  Print_drv() ==
  {
    Init();
    while true do
      ||(Write(),Error())
  }

  Init() ==
  {
    out_byte(Control_Port,
             init_printer_Command);
    out_byte(Control_Port,
             select_Command);
    put_irq_handler(IRQ,
                    print_handler)
  }
  ext rw status
  post status = online_Status

  Write() ==
  {
    if (data_count = 0)
      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

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

VDM-SLには16進数のデータ型は存在しないので,ここでは分かりやすさのため"0x"をプレフィックスすることにより16進数を表している.上記の仕様にはいくつか記述していない関数・操作および変数や,一部にVDM-SLの構文構造からは曖昧になっている表記が存在するが,ここでは詳しく記述しない.

3.2.2 LOTOS

以下はLOTOSでの記述例である(言語の詳細については文献[3]を参照).

```
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 = -6;
endtype
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
```

上記において，hex は 16 進表現のソートとして用いている．いくつかのゲート，オペレーション，プロセスなどをここでは省略している．

4 議論および考察

VDM-SL では，関数として副作用を伴わない仕様を記述し，操作として状態を変化させるような仕様を記述する．状態は大域変数を用いて記述するため，

ポート，ステータス，コマンドの値を大域変数に代入しておくことにより，どの操作からも利用する事ができて便利である．プロセスの実行は既存のプログラミング言語と同様に関数・操作を呼び出す事によって行なうため，プロセスの時間的な順序関係は呼び出された関数・操作の順番によって記述する事になる．

LOTOSには抽象データ型しか存在しないので，定数は演算から作らなければならない．値の引渡しはプロセス間の相互作用として行なわれるため，プログラミング言語の状態変数に相当するものを作ることと，変数のスコープを広くとることは難しい．プロセスの時間的な順序関係は，プロセスをイベントとみなすことによりLOTOSのサポートしている構文を使えば簡単に記述する事ができる．また，LOTOSでプロセスインスタンシエーションを使えば，VDM-SLの関数・操作に似た記述をする事ができる．

2つの言語ともにデバイスの仕様を記述する事が可能と考えられる．データ型はVDM-SLの方が扱いやすく，プロセス間の関係はLOTOSの方が記述しやすい．結果として，ポート番号などの定数を宣言しておける機能と，プロセスの時間的な順序関係を記述できる機能とを共に備えている言語であれば，デバイスの仕様を記述する事はかなり容易になる事が分かる．

5 おわりに

今回は，二つの仕様記述言語VDM-SLとLOTOSを用いて，プリンタデバイスの仕様を記述した．また，デバイスの仕様の記述に必要な言語の機能を考察した．

VDM-SLおよびLOTOSで書かれた仕様から，実際のデバイスドライバを生成するシステムを構築する場合には，記述の抽象化のレベルをどの程度にするかによって，どちらの記述がシステムを構築しやすいかが変化する．

実際にデバイスの状態を変化させる場合，LOTOSはイベントとして記述できるが，VDM-SLは関数・操作として記述しなければならないため，抽象度はLOTOSの方が上である．

今後の課題として，抽象度を下げた記述が挙げられる．すなわち，記述言語にVDM-SLを用いてデバイスドライバを生成するシステムを作って行く場合に，VDM-SLで書かれた抽象度を下げたデバイスの仕様記述から，デバイスの違いをどれだけ吸収できるかが課題となる．VDM-SLで生成システムを構築した後，LOTOSで抽象度を上げたデバイスの仕様の記述を用い，生成システムで行なわなければならない処理をVDM-SLで作成した生成システムに付加する形で，デバイスごとに明確に定義していく必要がある．

参考文献

[1] Xiaohua Jia and Mamoru Maekawa: *Operating System Kernel Automatic Construction*, Operating Systems Review, Vol.29, No3, pp.91–96, 1995.

[2] Cliff B.Jones: *Systematic Software Development using VDM*, Prentice Hall, 1990.

[3] 高橋 薫, 神長 裕明: 仕様記述言語 *LOTOS*, カットシステム, 1995.

[4] John Fitzgerald, Mercel Verhoef and Takanori Ugai: *VDMのFAQ(VDM-FAQ-JAP)*, 1994.

[5] Andrew S. Tanenbaum: *MINIX* オペレーティングシステム, アスキー出版局, 1989.

[6] The VDM-SL Tool Group and The Institute of Applied Computer Science: *The IFAD VDM-SL Language*, 1996.

[7] 水野 忠則監修: プロトコル言語, カットシステム, 1994.