デバイスとそのドライバを記述するための言語

2ITL: Logic which has process as a value of a variable

河野 真治　　安里 朋之
Shinji Kono kono@ie.u-ryukyu.ac.jp Tomoyuki Asato tom@cr.ie.u-ryukyu.ac.jp

琉球大学 情報工学科
Information Engineering, Univesity of the Ryukyus

概 要

計算機には、さまざまなハードウェアが拡張ボードやPCMCIAカードなどのような形で接続される。ハードウェア自身は、Verilogなどのハードウェア記述言語によって記述される。OSは、接続された機器をを直接理解することはできず、デバイスドライバをプログラミング言語で記述することになる。この論文では、この二つの記述のギャップによる不都合を解決する手段として、ハードウェアの記述とOS側のドライバを統一して記述できるような言語を提案する。この言語は、状態遷移機械と、従来のプログラミング言語的な記述の自然な変換を実現する。

1 abstarct:

計算機には、さまざまなハードウェアが拡張ボードやPCMCIAカードなどのような形で接続される。ハードウェア自身は、Verilogなどのハードウェア記述言語によって記述される。OSは、接続された機器をを直接理解することはできず、デバイスドライバをプログラミング言語で記述することになる。この論文では、この二つの記述のギャップによる不都合を解決する手段として、ハードウェアの記述とOS側のドライバを統一して記述できるような言語を提案する。この言語は、状態遷移機械と、従来のプログラミング言語的な記述の自然な変換を実現する。

2 何故、ドライバを記述するための言語なのか

世の中のOSや、プロトコルは、何か一つに統一されるかと思うと、その周辺では独自の拡張が激しく行われる。特に、コンピュータの周辺機器は毎年新しい技術が導入される。ハードの開発自身は、ハードウェア記述言語によりスピードアップが図られているが、最終的には、さまざまなOSや機器に対してのデバイスドライバを記述する必要がある。この部分は、対応するものによってアセンブラで記述されたり、Cで記述されたりすることになる。

もちろん、OSが一つに統一されれば、ただ一つのドライバを記述すれば良いわけだし、OSにはそういう役割を期待されていたところがある。しかし、現実には、電子手帳から超並列サーバまでを一つのOSが使われることはありえない。したがって、複数のOSと複数の周辺装置を結ぶなんらかの機構が必要であると考えられる。実際のコンピュータでは、OSは厳密な中間階層になっているわけではなく、数あるハードウェア、ソフトウェアの中の一つの部品になっている。その中で、ハードウェアとソフトウェアを対にして記述できる言語が必要だと考えられる。(図1)

本論文では、このための記述言語について考察する。

3 デイバスドライバ記述言語に対する要求

周辺機器とOSとのインターフェースの記述は、視点によって様々な要求がある。

単なる形式的な記述でしかない。つまり、この記述言語は、OSのAPIの動作を含めた記述が可能であることが望ましい。

また、可能ならば、デバイスドライバは自動生成されるべきである。さらに、十分な情報公開がある場合には、それに合わせた段階的改良が可能であるべきである。

また、デバイスドライバの記述は必ずハードウェアとソフトウェアの接点になる。したがって、ハードウェア記述言語と、ソフトウェア記述言語の橋渡しをする言語である必要がる。従来のハードウェアとソフトウェアの協調設計は、ハードウェア設計の視点から行なわれることが多くかった。例えば、組み込み機器などが目標であった。しかし、現実には、特定のアプリケーションに対してハードウェアを設計するよりは、特定のOSに対ししてハードウェアは設計される。例えば、Graphics Acceralator などがそれに当たる。これらは、例えば、Windows, Display PostScritpt, X-Window などに特化したドライバを同時に作る必要がある。

図1　ハードとソフトの部品化

OSから見て　周辺機器の仕様記述
周辺機器から見て　OSのインタフェース
ユーザから見て　OSを通して操作できる周辺機器の特別な機能

これらは、一つの技術公開の方法である。

この技術公開は、企業秘密に属するものである。従って、公開されない場合もある。そのような場合は、広く存在するOSに対するサポートは限られたものとなる。また、技術的なサポートができないことを理由に特定のOSに対するサポートをしないこともありえる。例えば、Windows 3.1で動作していた周辺機器が、Windows NT では動かないというような場合である。従って、形式的な方法で技術隠蔽を行ないつつ、段階的な仕様公開を行なうような方法が望ましい。

ある周辺機器、例えば、LSIの機能は、その電気的な性質やコマンドの名前だけから決まるわけではない。また、その機能を実現する詳細なハードウェアの記述がわかっても十分ではない。実際には、そのLSIがそれに関わるOSとどう関わるかを理解しない限りドライバを記述することはできない。実際、SCSI ポートの仕様は、そのreadやwriteの機能がOSの機能とどう結びつくかを記述しない限り

4　ハードウェア記述言語とシステム記述言語の関係

ハードウェア記述言語は、Verilog, VHDL などが知られているが、それらは、すべて有限状態遷移機械(FSM)と論理回路を対象にしている。一方で、システム記述言語はレジスタとスタックを前提とした記述になっている。それぞれは、まったく異なる設計手法が用いられるのが普通であり、記述量もシステム記述言語の方がはるかに多いのが普通である。

システム記述言語はハードウェア依存性を隠すために設計されているが、ハードウェア記述言語には、そういう概念はない。しかし、階層的な設計はハードウェア記述にもあるので、情報隠蔽という概念は両方に存在する。

OS内部の記述や、デバイスドライバの記述では、必ずハードウェア依存の記述を行なわなければならない。ここは、さまざまなトリックが使われる。例えば、アセンブラを含めた記述などが使われる。ハードウェア記述言語の方には通常はシステム記述言語とのインタフェースは存在しない。しかし、メモリ上のデータを特定のレジスタにマップしたり、

FSMによって特定のプロトコルで読み込むなどの設計がそれに相当する。例えば、特定のアセンブラコードの動作はハードウェア記述言語を使って記述することができる。

簡単に言えば、ハードウェア記述とソフトウェア記述の差は、スタックマシンとFSMの差に過ぎないということできる。従って、この差を吸収できる言語を設計することがデバイスドライバ記述言語の一つの目標になる。

5 継続と環境を制御から分離する

スタックマシンとFSMを結びつけるには、スタックの部分を分離してやれば良い。これは、通常のスタックベースの言語(CやPascal、今の言語は、ほとんどがスタックベースである)を実装する時に使われるスタック、フレームポインタ、戻り番地を分離すれば良い。

これらは、環境と継続(Environment and Continuation)を制御から分離する、あるいは、明に記述するということである。このようにすると、通常のスタックベースの言語と異なり、アセンブラの能力そのままを記述できるようになる。つまり、フレームポインタを通した操作は環境の操作であり、戻り番地に対する操作は継続に対する操作となる。

また、環境と継続を取り除くと、FSMを直接に記述できるので、この言語自身をハードウェア記述として使うことができる。従来のシステム記述言語では、巨大なgoto文またはcase文によってか、あるいは、関数呼び出しを使ったFSMの模倣しかできない。

6 継続と環境の型付け

継続と環境を直接扱うには二通りの方法がある。一つは、それらをfirst class object として定義してしまうことである。このようにすると、その内部構造は完全に隠されて、それに対する操作だけを定義することになる。Scheme などのLisp系の言語では、これらの方法が使われている。もう一つは、仮想的な実行系を考えて、その中での継続と環境を操作する方法を定義することである。これは、Reflectionと呼ばれる。

前者の方法は、プログラミング言語としてはすぐれているが、ハードウェア言語との橋渡しとしての機能はない。後者の方法は、Reflectionを直接実現することはできず、メタインタプリタまたは、前もって部分評価のような手法でコンパイルしておく必要がある。

ここでは、プログラミングを初めから、継続と環境に対する操作を明示的に記述することにする。そのために、継続と環境のデータ型を最初から記述する。制御が分離されているために、継続と環境のデータ型は普通のデータ型と変わらない。

通常のシステム記述言語は、継続と環境に対する操作を隠しているだけなので、デバイスドライバ記述言語へは、それを明示的にするだけで変換することができる。

7 セグメントとイベント

FSMの記述は、Cに近い構文で表現する。状態には、関数としての名前が付くが、その呼び出しは、状態遷移に過ぎず、引き数の積み込みやスタックの管理、戻り番地の格納はすべて明示的に行なう。状態遷移の中では、関数呼び出しを除くCの機能を使うことができる。この状態遷移をセグメントと呼ぶ。セグメントは関数呼び出しで結びつけられるが、特定の変数の待ち合わせを行なうこともできる。これをイベントという。また、関数呼び出しがない代わりに、状態遷移(goto)と、マクロ展開に相当するinlineがある。

イベントの制御には、if文と待ち合わせを行なうwhenなどのいくつかの原始文を用意する。

```
struct env_segment1 {
   void *fp;
   void *sp;
}

struct cont_segment1 {
   int value;
   *return();
}

segment1(env_segment1 e,
        cont_segment1 c) {
    cont_segment1 c1 =
      new_continuation(c,e);
    c.return = segment2;
```

```
    call_segment(e,c1);
}

call_segment(env_segment1 e,
       cont_segment1 c) {
    c.value = 1;
    c.return();
}

segment2(env_segment2 e,
      cont_segment1 c) {
    cont_segment1 c1 =
      old_continuation(e);
    halt(e,c);
}

inline type new_continuation(c,e) {
    e.fp -= sizeof(type);
    return (type)e.fp;
}

inline type old_continuation(c,e) {
    e.fp += sizeof(c);
    return (type)e.fp;
}
```

8 超低レベル記述

この段階での記述は、ほぼハードウェア記述言語の機能であり、記述のレベルは一番低い。例えば、式の途中で関数呼び出しを行なうようなものでも、呼び出す前と呼び出した後の二つのセグメントを記述する必要がある。これは、ハードウェア記述では、むしろ当然である。

しかし、普通のアルゴリズムを記述する場合は、セグメントの数が多くなり簡易な記述をすることができない。この場合は、例えば、CやPascalなどから、機械的にコンパイルすることができる。この方法は、ABCL/1などの実装でも行なわれている。このコンパイルは比較的自明である。

関数呼び出し等は継続により処理される。この時、システム全体での継続はたかだか有限個しかない。従って、この言語で記述される系は、環境の変化を無視すればFSMとなる。

また、継続を明示的に扱っているため、従来はアセンブラなどで記述していたコンテキストなどを記述することもできる。

9 より高度な記述へ

セグメントと、その環境と継続の型は、いくつかのまとまりにわけることができる。この分け方は、例えば、モジュールとしてのまとまりや、オブジェクトとしてのまとまりを使うことができる。しかし、実際の実行はセグメントにより起こる。従って、分け方は、プログラムが動作した後に、何種類かの方法で任意にわけることができる。

10 ハードウェアとのマッピング

通常のプログラミング言語の実行は、言語をアセンブラに変換すれば良い。しかし、デバイス記述言語では、フレームポインタやスタック、継続までを明示的に記述するために、単なるアセンブラによる実行よりも、厳密な対応を持つコンパイルが必要になる。任意の記述が効率よくハードウェアにマッピングされるわけではない。従って、デバイス記述言語のレベルでの最適化が必要になる。

この意味で、デバイス記述言語のレベルでの移植性・可搬性はむしろ低くなる。この言語では、移植性・可搬性は、インタフェースレベルで取るのであって、バイナリレベルでとるのではない。

11 まとめ

従来、ハードウェアとソフトウェアの協調設計は、通常のプログラミング言語とハードウェア記述言語の同時設計によっておこなわれてきた。ここでは、プログラミング言語の記述のレベルを意図的に下げて、よりハードウェアに近いレベルまで記述できるようにした。これにより、ハードウェア記述とソフトウェア記述を同時に行なうことができる。また、この記述を、モジュールに分割してインタフェースの部分だけを公開することによりAPIの記述をおこなうことができる。

より状態遷移に近い記述をおこなうため、Object指向開発とも相性が良いと考えられる。

状態遷移記述によるソフトウェア記述としては、Liberoという言語が知られているが、本言語では、継続を明示することにより、従来のプログラミング

言語からのコンパイルを可能にしているところが異　なる。