継続を基本とするプログラム単位を用いた組込みシステム開発

Embedded System Development using Continuation based Programming Unit

河野 真治†
Shinji KONO

† 琉球大学情報工学科,
Information Engineering, University of the Ryukyus,
PRESTO, Japan Science and Technology Corporation
kono@ie.u-ryukyu.ac.jp

近年の組込みシステム開発には、プログラミング言語アセンブラ言語ハードウェア記述言語などのさまざまな言語が必要となる。様々なプログラム/ハードウェア/仕様/制約の記述に応じて様々な言語が開発されて来ているが、それは、一方で、開発者の教育的なコストを大きくし、負担になっている。そこでは、我々の開発した「継続を基本とした新しいプログラム単位 (code segment)」を実現する言語処理系 CbC (Continuation based C) [4] 開発した。これを、アセンブラ、資源制約記述、メタレベル、シミュレータ、仕様記述、組込み機器の検証システム、などに使用する C と互換性のある記述言語として使用する方法に付いて考察する。

1 組込みシステム開発

近年の組込みシステム開発には、さまざまなツールが使われている。それぞれについて、さまざまな言語が必要となる。(fig.1)

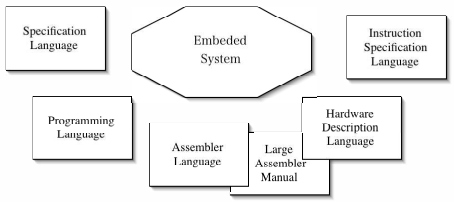

Figure 1: さまざまな言語

また、これらは必要に応じて組込みシステム特有の拡張が存在することが多い。例えば、組込み機器専用のライブラリはプログラミング言語に対する拡張であり、ハードウェアに対応した特殊な命令の拡張や、特殊なメモリアドレスまたは I/O ポートへのアクセスはアセンブラ言語への拡張となる。ハードウェア記述言語に対しても低消費電力や耐久性、発熱などの資源制約は言語的なアノテーションとして拡張されることが多い。

また、組込みシステム特有の資源制約、あるいは、仕様記述は、なんらかの論理式によって記述されることが多い。例えば時相論理や Z などによる記述が使われる。モデル検査などの手法が使われる場合は、その仕様記述は CTL などの論理式によって記述される。

これらは、さらに、RTOS (Real-time Operating System) や、Java や C 言語などのアプリケーションと結び付いて実行されることもある。携帯電話の i アプリや、PDA、ゲーム機などは、そのようなシステムとなる。

RTOS へのアクセスや省電力のためのハードウェア操作 (クロックの制御や部分的な回路の停止) などは、システム記述から見ると、一段メタなレベルのシステムに対する操作記述であり、これらに対する記述をリフレクションやアスペクト指向プログラミングで記述するというアプローチもある。これらは、プログラム言語の拡張として導入されることが多い。

これらの目的のために、様々なプログラム/ハードウェア/仕様/制約の記述に応じて様々な言語が開発されて来ているが、それは、一方で、開発者の教育的なコストを大きくし、負担になっている。非常に強力な表現力を持つ言語と、そのシステムがあっても、それを学ぶのに半年かかるのでは、現実の組込みシステム開発に使用されることはない。

ここでは、我々の開発した「継続を基本とした新しいプログラム単位 (code segment)」を実現する言語処理系 CbC (Continuation based C) [4] を紹介し、それらを、

アセンブラ
資源制約記述
メタレベル
シミュレータ
仕様記述
組込み機器の検証システム

などに使用する方法に付いて考察する。

2 継続を基本とした新しいプログラム単位 (code segment)

ここで導入する単位はステートメントよりも大きくサブルーチンよりも小さい単位である。ループ構造を持たないステートメントの集合であり、コンパイラでは基本ブロックと呼ばれるような部分に相当する。また、並列実行やスレッドなどの実行をプログラムの分割によって疑似的に実現する場合にも使われて来た単位である。これをプログラム単位としてプログラム言語的に明示的に使用するのが、ここでの新しい提案である。この単位を code segment と呼ぶ。code segment は継続 (continuation) で接続される。継続への移動は引数付きの goto 文 (parameterized goto statement) で表現される。code segment 自体は制限された C のステートメントにより表現される。この言語を CbC (Continuation based C) と呼ぶ。

継続を持つ C に近い言語としては、C - - [3] が知られているが、CbC は、継続を基本とするところが異なる。

code segment は入力 interface から条件文によって分岐する複数の出力 interface を接続する単位となっている。状態遷移系を直接に表現する単位となっている。(fig.2)

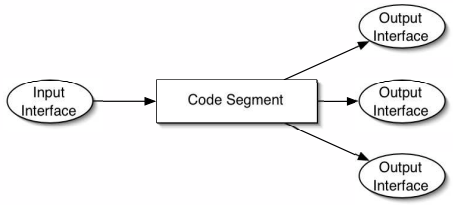

Figure 2: code segment

以下の例は、CbC による階乗の計算である。

```
code fact(int n,int result,
       code (*print)()){
    if(n>0){
        result *= n;
        n--;
        goto fact(n,result,print);
    } else
        goto (*print)(result);
}
```

`goto fact(n,result,print);` は、直接の継続であり、その引数は、interface と呼ばれる。属する `code` と同じ interface を持つ `goto` 文は、一つの jump 命令にコンパイルされる。

`goto (*print)(result);` 間接の継続である。CbC の継続は Scheme などの継続と異なり環境を切替えない。これを軽量継続 (light weight continuation) 呼ぶ。

スタックを自分で記述することにより、通常の C 言語で記述されたプログラムを CbC で表現することが可能である。従って、CbC は、C の下位言語であると言える。

C 言語に対して、code segment と 引数付き goto 文を付加する言語を定義するためには、C のサブルーチンへ戻るための環境付き継続を導入する必要がある。この言語を CwC (C with Continuation) と呼ぶ。CwC は C の上位言語である。CbC は、CwC の仕様の一部に制限されたものとみなすことができるので、CwC を CbC として使うことも可能である。

CwC は現在、IA32 用、PowerPC 用のコンパイラが実装されている。

3 アセンブラとしての使用法

アセンブラは、CPU が実行する命令のビットパターンを人間が読める形にしたものである。通常は、命令のテキスト形式 (命令部分であるオペコードと可変引数であるオペランドからなることが多い) と、ビットパターンの対応、そして、命令の動作の詳細を記述した巨大なマニュアル (数百頁) が存在する。命令動作は、CPU 内部のレジスタやフラグに対する動作として記述される。これらの命令は CPU のハードウェア記述によって決まるが、ISP などの専用の記述言語も存在する。

CbC では、アセンブラの動作の記述を以下のように記述する。

```
code opcode(struct cpu_register reg,
            struct operand opr,code next()) {
    reg.eax = opr.value;
    reg.pc += 2;
    goto next(reg);
}
```

これは動作記述なので、interface に含まれる `cpu_register` が実際の CPU のレジスタに対応するかどうかは重要ではない。

この code segment を呼び出す場合は継続として、次のオペコードとオペランドを取り出し、それに対応する動作記述 code segment を呼び出す code segment を指定する。これは、ハードウェア記述ではデコーダに相当する。

```
code decode(struct cpu_register reg) {
    code opcode();
    opcode = opcode_table[mem[reg.pc]];
    goto decode_operand(opcode,reg);
}
code decode_operand(opcode,
       struct cpu_register reg) {
    struct oprand opr;
    opr = oprand_table[mem[cpu_register.pc+1]];
    goto opcode(reg,opr,decode);
}
```

これは CbC を ISP のように命令動作記述として使うことに相当する。この場合はアセンブラはビットパターンとして配列 `mem` に格納されることになる。(fig.3)

実際、PS2 の VU のアセンブラの動作記述をこの方法により記述することができた。[5]

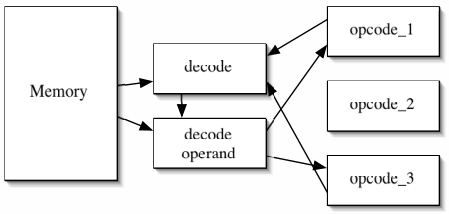

Figure 3: 動作記述に近い記述

4　よりアセンブラ言語に近い使用法

アセンブラは、レジスタに対する操作の列と、その列の間の移動の記述である。これらを、配列に格納するのではなく、CbC で直接に記述することもできる。

```
code command_00001(struct cpu_register reg,
        struct oprand opr) {
    if (reg.eax == opr.value)
goto command_00010(reg);
    else
goto command_00002(reg,opr);
}
code command_00002(struct cpu_register reg,
        struct oprand opr) {
   ...
}
```

この場合はプログラムカウンタは明示的には表現されない。最後の goto 文が次の code segment を指すときには省略可能であるとすると少し短い記述が可能になる。(fig.4)

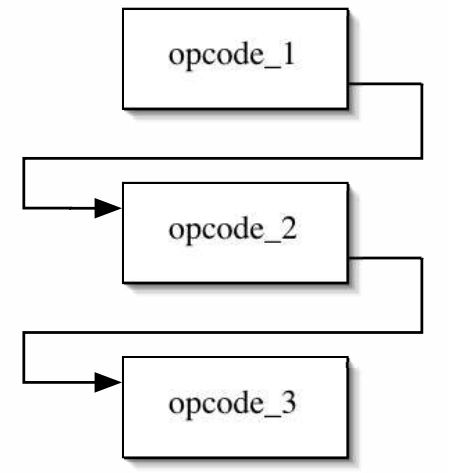

Figure 4: アセンブラに近い記述

それぞれの command_00001 の動作記述が命令の動作記述と一致する場合は、そのコマンドに対応する命令が一意に決まる。

一意に決まらない場合でも、レジスタに対する操作を実現する命令を探して当てはめるコンパイラを作成することが出来る。コマンド内部のステートメントを制限することにより、そのようなコンパイラを比較的容易に作成することが出来る。このようにすると、

```
code label_000(struct cpu_register reg) {
    reg.ecx --;
    reg.eax = *reg.esi;
    if (reg.ecx == 0)
goto label_001(reg);
    *reg.edi = reg.eax;
    reg.edi ++;
    reg.sdi ++;
    goto label_000;
}
code label_001(struct cpu_register reg)
   ...
}
```

のような記述が可能になる。これは、 Z80 などのアセンブラで採用されたような natural assembler に近い記述である。このプログラムは、もちろん、CbC の言語処理系により実行可能な記述であり、コンパイラにより機械語にも落せる記述でもある。

ハードウェアの拡張や CPU のバージョンアップなどにより命令が増えた場合には、command_0001 などのような記述を使用して対応することが可能となる。

この記述は機械語と CbC の記述の対応が 1 対 1 にならない。しかし、動作が良くわかるという利点がある。また、アセンブラの代わりにコンパイラを作成する必要がある。しかし、CPU 毎に異なるアセンブラ言語 (と、そのマニュアル) を作成する必要がないという利点がある。これは、アセンブラを習得するというコストを考えると大きな利点と思われる。

5　古いソースコードの解析ツールとして

アセンブラあるいは C、さらに古い言語 (例えば、PL/M など) で書かれた古いソースコードを繰り返し利用することは、組み込みシステム開発では良くある。しかし、古い部分に対して使えるツールが古い処理系ことによる制約、例えば、Z80 などのバイナリを生成する PL/M コンパイラしかないなどにより、新しいデバイス (CPU など) が使えないことは、システム全体の質を下げてしまうことになる。

システムを構成する技術は日進月歩で進歩している。それに対応する古いソフトウェアの部分、、あるいは、取り残されたハードウェア部分を、新しい技術に対応させることが必要である。これは、既存のプログラムやハードウェア設計を新しい技術に対応させる問題であり、テクノロジーマッピングの問題である。つまり、既存のソフトウェアやハードウェア設計を、新しい CPU やハードウェア設計に対応させることが必要となる。そのためには、既存の設計やハードウェアの動作を分析し、エミュレーションあるいは、新しいハードウェア上での実行に置き換えることが必要である。

CbC は、C からコンパイルすることができるのと同様に、アセンブラあるいは古い言語処理系から比較的簡単に移行することが出来る言語である。そ

れは、アセンブラや他の中間言語 (例えば p-code や Java VM) に比べて、抽象度と記述力が高いためである。

6 C や他の言語からの変換

C からの変換はコンパイラによって行なわれるが、変換の中心は、スタックの明示的な記述である。

```
j = g(i+3);
```

のような関数呼び出しは、

```
sp -= sizeof(
   struct f_g0_save);
c = sp;
c->ret = f_g1;
goto g(i+3,sp);
```

のような形で、明示的なスタック操作に変換される。`f_g1` は、関数呼び出しの後の継続であり、`g` では、

```
code g(int i,stack sp) {
    goto (* ((struct
     cont_save *)sp)->ret)
       (i+4,sp);
}
```

のように間接的に呼び出される。スタックの中は、継続と中間変数などを格納する構造体である。スタックそのものは、これらの構造体を格納するメモリである。このスタックはリンクトリストを用いて実装しても良いが、ここでは、C のキャストを用いて型のないメモリとして実装している。

他のプログラム言語から変換する場合は、一度、C に変換し、そこから C から CbC への変換系をする方が実用的であると思われる。

再帰呼び出しのない場合は、関数ごとに中間変数と継続を格納する C でいう static 領域を設けることにより、スタックなしでの変換を行なうことが出来る。これは、FORTRAN などで用いられて来た技術である。組み込みシステムでは再帰呼び出しが使われることは希なので、その方がより明確な実装方法となる。これにより、スタックの使用量などを見積もる必要がない。

7 資源制約記述またはメタレベル記述としての使用法

組込みシステムの大きな特徴の一つとして資源を意識したプログラムがある。資源としては、

メモリ
I/O デバイス
CPU クロック
乗算器、加算器
電力
温度

などがある。これらは、動作記述には通常は現れない。また、実際のプログラミングでも明示的には扱われない。

これらを CbC レベルで表現するためには、資源を表す引数を導入する。例えば、クロックを表現するためには、

```
code label_000(struct cpu_register reg,
  struct cpu cpu) {
    cpu.clock += 3;
    reg.ecx --;
```

などのように行なう。I/O デバイスや乗算器などの資源の管理もこのような追加の引数で表現することが出来る。

これらの追加の引数はオブジェクトレベルの記述に対してメタレベルの記述と言うことが出来る。メタレベルの引数に対する操作はメタレベルへの操作となる。これらの引数と、その処理は、プログラム変換などで付加することが可能である。

このような記述で可能になるのはシミュレーションや検証であり、これが実際の資源の使用状況と一致する保証はないがシミュレーションの範囲内で一致する用途では使用できる。

メタレベルへのアクセスは、リフレクションと呼ばれる。さらに、メタレベルのデータをオブジェクトレベルに導入するレイフィケーションと、そのデータへの変更をメタレベルに反映するリフレクションの二つに分離して記述される場合もある。CbC の場合は実行に関与する明示的でない環境 (コールスタックや局所変数フレーム) などが存在しないために、オブジェクトレベルの操作とメタレベルの操作を区別するものは必要ない。従って、

```
code segment(object level variable,
            meta level variable) {
    meta level operation;
    object level operation;
}
```

のように、どちらも同じ記述で処理することが出来る。(fig.5)

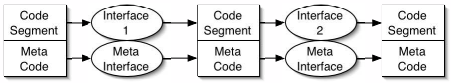

Figure 5: メタレベルの記述

リフレクションの例として良く使われるのは大域脱出であるが、スタックを明示的に記述する CbC では、大域脱出用の継続を表す引数と、その呼び出しで表現される。

```
code fact(int n,int result,
       code (*print)(),code (*exit)()){
    if(n>0){
```

```
        result *= n;
        n--;
        goto fact(n,result,print);
    } else if(n<0){
        goto (*exit)(-1);
    } else
        goto (*print)(result);
}
```

このような記述は関数呼び出しを持つ言語では不可能である。CwC では関数呼び出しが存在するが、code segment から C の関数に大域脱出的に戻るためには明示的な環境を用いる必要がある。CwC では、

```
goto (*ret)(value),env;
```

のような環境付きの goto 文が用意されている。

8　スレッドと CbC の記述

単一 CPU 上でのスレッドの実行は、基本的にはコルーチンとなる。CbC は自分の計算の環境を引数としてすべて明示的に持ち歩いているために、スケジューラを自分自身で記述できる。

```
code wait_thread_a(interface a self,
      scheduler s) {
    code wake_thread_a(interface a,
       scheduler s);
    extern int volatile message *msg;
    if (msg) {
        a->message = msg;
        goto wake_thread_a(interface a,
          scheduler s);
    } else {
        a->continuation = wake_thread_a;
        (interface a)(s->current) = a;
        goto s->next_thread(s);
    }
}
```

関数呼び出し機構を隠蔽している言語ではスケジューラを記述するには、なんらかのアセンブラによるサポートが必要である。これは大域脱出が CbC では自明に記述できることに対応している。従って、単一 CPU 上の RTOS などは CbC 自体で記述することが可能である。

複数 CPU の持つ場合は、資源の管理の問題があるので、状況は若干複雑になる。複数の code segment が同時に共有したメモリにアクセスすると、その結果は保証されない。保証するためには、なんらかの同期機構または排他制御が必要である。同期機構の単位としては、code segment 単位になることが自然だと考えられる。その場合、複数 CPU 上のスレッドの動作記述は、スケジューラの記述を使用することにより、CbC により記述することが出来る。従って、複数 CPU 上のスレッドの CbC による動作記述と、CbC による実装記述は異なるものである。(fig.6)

排他制御を実現する実装記述は、例えば、lock prefix を持つアセンブラ命令や、test and set 命令、あるいは OS のシステムコールに、コンパイルされる必要がある。一方で、動作記述は、スケジューラを用いたメタレベル記述で行なわれる。実装記述はアーキテクチャ依存であるが、動作記述はアーキテクチャに依存しない。あるいは、動作記述はアーキテクチャ依存部分を明示的に記述した記述となっている。

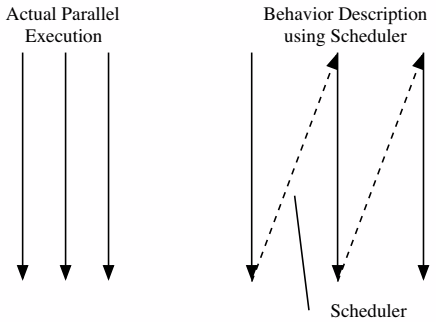

Figure 6: スレッドの記述

code segment を単位としない同期機構を採用する場合は、CbC は、その動作記述を code segment level で提供することはできない。動作記述をしたい場合は、記述をより詳細化し、同期機構の単位と code segment の単位を一致させる必要がある。この場合、code segment は、本質的に single thread な実行意味を持つことになる。

動作記述では code segment が実際に同時に実行されるかどうかは、メタレベルの変数に割り当てられた状態が表現するだけである。実装では、スレッドやプロセス、または、ハードウェア上で並列実行が行なわれる。

9　ハードウェア記述として

前節の CPU の命令の動作記述は、ハードウェア記述とみなすことも出来る。ただし、CbC では、引数または interface と、ハードウェアの対応を直接的には規定していない。これらの対応を決めるのは、ハードウェア・コンパイラの役割である。

ハードウェア記述として CbC を用いるためには interface がハードウェアに対応する必要がある。つまり、それらは決まった配線か有限なメモリまたはレジスタに対応する必要がある。クロックなどの記述は資源の記述としてメタレベル (つまり追加の引数として) 記述する。

ハードウェアとソフトウェアの切り分けは、なんらかの API を通して行なわれる。例えば、

特殊なアセンブラ命令
特殊なポートまたはメモリへのアクセス

がハードウェアとソフトウェアの接点となる。ハードウェアとソフトウェアを同時に記述する場合は、これらの接点は、code segment を単位としてとして記述される必要がある。

ハードウェア記述は、なんらかのクロックを元にした記述か、データフローの記述である。一方で、ソ

フトウェアの記述は、単一スレッドか複数スレッドでの記述となる。ハードウェアとソフトウェアの記述の分離は自明なものではなく、前もって、どのような実行をするかを考える必要がある。

元のアプリケーションの記述全体を、例えば、Cで記述し、それをCbCに変換することもできる。変換された結果はスタックやシステムコールを持っているので、その部分を含む部分と、そうでない部分に分離する必要がある。スタックを含まない部分は、自明にハードウェア記述として用いることができる。スタックが含まれている場合は、その部分を外部のメモリとして実装する必要があり、code segmentの継続が含まれている場合は、その継続のメモリ上でのエンコードを決定する必要がある。ハードウェア記述部分は、クロックに従って常に実行される部分なのか、シーケンサーに従って順序良く実行される部分なのかを分離する必要がある。これらの記述はCbCでは、同じようなcode segmentによって記述されてしまう。これは、統一的な実装記述を提供するという意味ではプラスだが、ハードウェアとソフトウェアの分離が自明ではないと言う点ではマイナスである。

10 シミュレータとしての使用法

CbCで記述された動作記述は、アーキテクチャに依存しない。CwCを用いるか、CbCプログラム変換を用いることにより、任意の時点での動作状態を調べることが出来る。メタレベルの状態を含む状態が得られる。もちろん、これをC言語や、System C [2] あるいはSpec C [1]でも行なうことが出来る。C言語の場合は、シミュレータを自分で記述することになる。System Cあるいは Spec Cの場合は、その複雑な操作意味論に従って動作することになる。CbCでは、複雑な動作意味論を規定する替わりに、状態遷移系を明示的に記述することにより、これを実現する。

CbCはアーキテクチャ依存性がないので、CPUを持つ組込み機器ならば、例えば、予定している拡張機能をCbCで記述し、その機器に組み込むことが可能である。もちろん、ハードウェアで実現する予定の拡張機能をCPUで実現するような場合は、速度的なペナルティは大きい。しかし、実機上での拡張機能のある程度の評価を行なうことができる。

また、実機がかなり遅い機器の場合でも、CbCでの記述そのものは高速なアーキテクチャ上で動作させることができるという利点もある。

CbCは、シミュレータを記述するための、言語とスケジューラなどのツールの集合を提供していると考えることも出来る。

11 仕様記述としての使用法

仕様記述としては一般的には時相論理などが使われることが多いが、$\Box (p \rightarrow \Diamond q)$ などの直観的な意味を把握することには向いているが、その精密な意味は、論理の詳細を理解しないとつかむことは出来ない。一方で、これらの仕様の意味はオートマトンとして定義することが可能である。従って、CbCによる状態遷移記述を時相論理的な仕様記述として用いることも可能である。

検証手法としては、実装記述とともに同時に仕様記述されたオートマトンを走らせるassert的な使い方が考えられる。また、interfaceの状態の状態を有限な状態に抽象化することができれば、実装記述とともにタブロー展開などの手法で直接的に実装の正しさを証明することも可能であると考えられる。

時相論理からオートマトンを生成する手法はいくつかあり、そのオートマトンの生成先としてCbCを使うこともできる。このオートマトンは直接に実行することができるという利点がある。

12 まとめ

ここでは、組み込みシステムのさまざまな段階で、継続を基本としたプログラミング単位の記述をどのように使うことが出来るかを考察した。

CbCは、C言語が理解できれば容易に理解できる構造を持つ。また、コールフレームなどの隠れた情報を持たないために、メタレベルの記述を容易に取り込むことが出来る。また、アセンブラとして使用すると複雑でアドホックな記述を使わずに機械語と動作記述を結び付けることが可能になる。

CbCは状態遷移系と相性が良いため、状態ベースの検証システムとの相性も良いと考えられる。

いくつかの使用例により、これらの特徴が組み込みシステム開発に向いていることを示した。

12.1 *References

[1] Daniel D. Gajski, Jianwen Zhu, Rainer Dmer, Andreas Gerstlauer, and Shuqing Zhao. *SPEC C:SPECIFICATION LANGUAGE AND METHODOLOGY*. KLUWER ACADEMIC PUBLISHERS, 1999.

[2] Stan Liao, Steve Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware. In *DAC '97*, 1997.

[3] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[4] Shinji Kono. A Continuation based Programming Language for Embedded Systems. In *IPSJ Computer System Symposium*, November 2000.

[5] 佐渡山 陽, 河野 真治 (琉球大) . Continuation Based C による PS2 Vector unit のシミュレーション . 情報処理学会システムソフトウェアとオペレーティング・システム研究会, June 2002.