ハードウェア記述言語SFLのオブジェクト指向化

吉田紀彦*・森木俊臣**
犬尾　武**

Object-Oriented Extension to Hardware Description Language SFL

by

Norihiko YOSHIDA*, Toshiomi MORIKI**

and Takeshi INUO**

The scale of VLSI's has been getting much larger. Current Hardware Description Languages (HDL) including SFL are procedural languages like Pascal or C, and hardware components cannot be custamized when reused. We are studying an object-oriented extension to HDL to promote hardware component reuse. This paper describes the research direction and basic concepts of object-oriented HDL (OO-HDL), investigates inheritance anomaly in hardware component inheritance, and proposes a solution. This paper also presents our language prototype SFL++ which is an extension of SFL.

1．はじめに

今日のVLSIの規模は従来の設計法で対応できるような範囲を越え，単なるツールや設計環境では，素子数の増加という確実な要因によってその役割を果たせなくなるのは目に見えている．そこで，ある程度抽象的な機能や動作の言語記述を設計対象とし，これをもとに論理シミュレーションを行なったり論理合成を行なわせる設計法が，今後もっとも有効でかつ効率的な手段として期待されている[1]．この設計方法は高級言語を用いるプログラム開発とほぼ同じ形であるが，現在のハードウェア記述言語（HDL）は，手続き的・関数的な処理を主体としており，部品化やその再利用はモジュールライブラリの段階に留まっている．

そこで本稿では，HDLにおける部品化の促進に向けて，そのオブジェクト指向化について述べる．オブジェクト指向化により，その代表的な特徴である，継承での差分プログラミングによる効率的なモジュールの記述，及びクラス階層によるハードウェア部品の分類などが容易に可能となる．さらに，変数を導入したパラメトライズド・モジュールを用いることで，より抽象度の高い部品の記述が可能となる．

また，ハードウェアは並行システムであるので，並行オブジェクト指向における継承異常の問題[2],[3]が発生する恐れがある．そこで，オブジェクト指向HDL（OO-HDL）における継承異常の問題も分析し，その解決策を示す．次いで，以上の提案の有効性を実地に検証すべく，代表的なHDLの中から同期式回路を対象としたSFL[4],[5]を取り上げ，これにオブジェクト指向の機構を取り入れて我々が設計開発を進めているOO-HDLであるSFL++について，その言語仕様と処理系の概要を述べる．

平成10年10月1日受理

*情報システム工学科（Department of Computer and Information Sciences）
**九州大学大学院システム情報科学研究科　福岡市東区箱崎（Graduate School of Information Science and Electrical Engineering, Kyushu University, Hakozaki, Fukuoka）

2．ハードウェア記述言語

2.1 概要

集積回路技術の進歩によって VLSI は大規模化が進んでいるが，一方でその設計支援に関して，ASIC（Application Specific IC）技術，および80年代後半での論理合成技術の実用的なレベルへの成長によって，ソフトウェアにおけるプログラミングと同じようなVLSI 設計が可能になりつつある．つまり，ソフトウェアでの高級言語に対応する HDL を用いることで，従来専門家が設計を行なっていた回路の設計を，一般の利用者が論理設計などの知識がなくても設計できるようになってきている．

HDL はもともとシミュレーションや性能評価といった目的のために考案された言語で，Ada の拡張として設計された IEEE 標準規格の VHDL，シミュレータ入力言語として開発された Verilog HDL，我国を中心に標準化が進行している UDL/I などがある．一方，VLSI 回路設計，すなわち論理合成においては，HDL の抽象的記述から，ネットリストと呼ばれる VLSI 回路のトポロジ（論理素子の種類とその接続関係をリストとして記したもの）を，人手を介さずに自動的に生成する技術が固まりつつある．その先駆けとなった HDL の１つとして SFL が挙げられる．

HDL によるハードウェア開発を高級言語によるソフトウェア開発と照らし合わせると，論理シミュレーションはインタプリタによる解釈実行に相当し，論理合成によるネットリスト導出はコンパイラによる実行モジュール生成に対応する．

2.2 SFL

SFL[4],[5]は NTT で開発された HDL であり，簡便で十分な機能の環境が整っており，現在，我国の教育機関でもっとも多く利用されている．

先に挙げた HDL のうち，VHDL や Verilog HDL は主にシミュレーションを対象としていて，論理合成のための記述は別のものとして区別されている．特に VHDL はアナログ回路まで対象としており，言語仕様が巨大で複雑である．UDL/I では，シミュレーション用と論理合成用の記述方式のギャップは比較的小さい．しかし，これら３つの言語はいずれも，動作記述から論理素子のトポロジまでの記述が可能になっており，次元の異なる記述レベルが混在している．

それに対し，SFL は効率的な設計を促すために提案された言語で，論理設計に関する部分と動作設計に関する部分がはっきりと区別されている．そして，対象を CPU などのディジタル回路の主流である同期式回路に限定することで，言語仕様をコンパクトなものにしている．記述は動作に関する部分だけでよく，論理設計は論理合成システムにより自動的に行なわれる．また，すべての動作を手続きで記述するため，配線に関する記述は一切必要ない．加えて，同一のソースコードでシミュレーションも論理合成も可能となっている．すなわち，SFL は動作だけを手続きのみを用いて記述する新しい設計方法であり，HDL の中ではもっともソフトウェアの高級言語に近いといえる．

そこで我々は一般の HDL のオブジェクト指向化の最初の叩き台として SFL を取り上げる．本稿での議論は他の HDL にも同様に適用可能である．

SFL はモジュール単位に記述し，モジュールの包含関係を階層構造で表している．動作に関しては，起動されたマシンサイクルのみ動作する手続き（組合せ回路に対応）と，１度起動されると複数のマシンサイクルに渡って状態遷移しつつ動作するステージによる手続き（順序回路に対応）が存在し，モジュールの最初の起動は制御端子によって指定される．また，SFL のモジュールの内部状態は，state と呼ばれる最小単位で表現され，stage と呼ばれる単位の内部で状態遷移を行う．stage は１つのモジュール内で複数存在する．SFL で記述した簡単なカウンタの例を以下に示す．

```
module counter {
 circuit_type incre {
  input in<10> ;
  output out<10> ;
  instrin up ;
  instr_arg up(in) ;
  instruct up out = in + 0b1 ;
 }
 input in<10>, flg ;
 output out<10> ;
 instrin start ;
 reg counter<10> ;
 incre inc ;
 stage_name cnt { task run(counter) ; }
 instruct start generate cnt.run(in) ;
 stage cnt {
  state_name st1,st2 ;
  first_state st1 ;
  state st1 par {
   counter := inc.up(counter).out ;
   out = counter ;
   any { flg : goto st2 ; }
```

```
    }
    state st2 par {
      counter := in ;
      goto st1 ;
    }
  }
}
```

3．並行オブジェクト指向

3.1 概要

まずオブジェクト指向化に先立ち，その種類と問題点を述べる．

オブジェクトには大きく2種類あり，1つはシステム全体が1つの逐次的な制御で実行される逐次オブジェクト．もう1つはシステムの各部分が並行に動作する，オブジェクトの持つ本来の意味通りにモデル化された並行オブジェクトである．さらに，並行オブジェクトはその内部の処理の制御の流れ（実行アクティビティー）の数により単一スレッド型と多重スレッド型に分類される．

逐次オブジェクトはメッセージの送信において，制御の流れも同時に送信し，返信とともに制御の流れも受けとる．並行オブジェクトでは，単一並行スレッド型の場合，メッセージ送信後も制御の流れを渡さずに，メッセージを受けとったオブジェクトで新たに制御の流れを生成し実行を始める．ただしオブジェクト内での制御の流れは多くて1つである．多重スレッド型の場合は，オブジェクト内で持ち得る制御の流れの数が複数まで可能となる．システム内での制御の流れの数が増せば，それだけ並列度が増すことになる．

これらのオブジェクトをSFLの構成要素に当てはめると，moduleは多重スレッド型，stageは単一スレッド型の並行オブジェクトに対応する．また，stateはオブジェクト内における1つの実行単位に相当するが，メッセージの流れに着目すれば逐次オブジェクトに分類される．

ただしソフトウェアでの並行オブジェクトとの違いを詳しく議論すると，SFLにおいてはすべてのメッセージはクロックに同期して行なわれ，しかも出来上がったハードウェアが静的なものであるため，モジュールに対応するオブジェクトを動的につくり出すことは不可能であり，実行アクティビティーの個数も静的に固定される．すなわち，ソフトウェアでは非同期にメッセージが届くことを意識してオブジェクトを構築する必要があったが，SFLではすべての動作をクロック単位に区切って記述でき，すべての振る舞いが静的であるため，ハードウェアではソフトウェアでの並行オブジェクトより単純なオブジェクトとなる．

3.2 継承異常

一般にオブジェクト指向，特に継承を導入することによって継承異常（Inheritance Anomaly）が発生する可能性がある．

継承異常とは，継承を行なう際に広範囲に渡ってカプセル化が破損してしまう現象のことであり，この問題により継承による部品の再利用性が低くなり，すっきりしたアプリケーションフレームワークの構築が困難になることがある．最悪の場合，スーパークラスのどのメソッド（処理手順）も継承できないような例が一般的に存在している．

ソフトウェアにおいて，この問題は重要であり，解決案がいくつか提案されている．現在もっとも有力な解決法としては，問題となるコードを明確に分類し，それぞれを区別し，独立して継承を行なうことでる．

例えば並行ソフトウェアでのプログラミングにおいては，1つのメソッドの中に大きく3つの種類のコードが存在する．それらは，実行中のアクティビティーを知るための同期に関するコード，オブジェクトの状態を知るための振る舞いに関するコード，そして実際の処理に関するコードである．

例えば，新しいメソッドを相互排除で追加したい場合には，すべてのメソッドの同期に関する部分を修正する必要がでてくる．つまり，振る舞いや実際の処理は継承前と同じであるにもかかわらず，継承することによってその部分を含めたメソッド全体を再び書き直す必要が発生する．すなわち継承による差分プログラミングの利点が全く生かされていないことになる．

そこで，これらのコードを区別し各々を継承することによって，各コードの再利用性を高めようというのが並行ソフトウェアでの継承異常の解決策である．

SFLに関する継承異常については次の章の中で紹介する．

4．SFLのオブジェクト指向化

4.1 基本概念

現在のHDLはいずれもハードウェア部品単位のモジュールの概念を持ち，カプセル化は可能であるが，再利用性に関しては，ソフトウェアにおけるモジュールライブラリの段階に留まっていて，継承によるカスタマイズなどはできない．

VHDLについては，オブジェクト指向化の動きが幾つか見られる．例えば，IEEE設計自動化規格委員

会ではオブジェクト指向 VHDL 部会をつくり，検討を始めている[6]．しかし，正式な提案はまだなされていない．また，米国 Vista Technologies 社が OO-VHDL[7]を発表しているが，これは VLSI 設計よりもハードウェアシステムのモデリングを主な目標としている．

このように，オブジェクト指向化に関する動きはあるが，我々が関心を持っている論理合成システムの存在を前提にした研究はまだ行なわれてはいない．そこで我々は，SFL にオブジェクト指向を取り入れ，その有用性を検証するべく，新しい言語（SFL++）の設計を行なっている．SFL は同期型回路のみを扱い，言語仕様がコンパクトであり，論理設計を自動化することで純粋に動作のみの記述に徹しているので，HDL の中では最も高級言語に近い言語であると言える．

SFL や VHDL などはすでにモジュールの概念を持っているので，オブジェクト指向化の中心となるのは継承システムの導入となる．

module をオブジェクトとして扱うと，module の動作の最初の起動は制御端子で指定されるので，これを一般のオブジェクト指向言語のメソッド呼出しと考えることができる．すなわちメソッドに対応するのは，module 内で定義されている制御端子が宣言であり，stage が実装となる．一方 state は stage 内で閉じた一連の状態遷移の連鎖なので，メソッドの一部と解釈できる．

よって，SFL のオブジェクト指向化においては，基本的に次の言語要素が継承の対象となる．

・構成要素（各端子，レジスタ，メモリ等）
・制御端子の動作
・stage

なお，これは VHDL のオブジェクト指向化でも同様であり，IEEE 内部案や OO-VHDL では operation（メソッドに相当）単位の継承が考えられている．

4.2 ハードウェア部品の継承異常

一般の並行オブジェクト指向システムでは，継承異常（Inheritance Anomaly）の問題[2],[3]が発生する．ハードウェアは互いに並列に動作する単体モジュールの集まりであるので，HDL が表現しているモデルは，状態遷移をスレッドと考えると，単一スレッド型並行オブジェクトと捉えられる．しかし，オブジェクト指向 HDL では，継承異常に相当する問題は，状態遷移の追加・上書きという，よりプリミティブなレベルで発生する．この例を以下に示す．これは先のカウンタにリセット機能を付け加え修正したものである．

```
module counter_with_reset {
 super counter ;
 input reset ;
 stage cnt{
  state_name st1,st2,st3 ;
  first_state st1 ;
  state st1 par{
   counter := inc.up(counter).out ;
   out = counter ;
    any { flg: goto st2 ;
          reset:goto st3 ; }
  }
  state st2 par{
   counter := in ;
   goto st1 ;
  }
  state st3 par{
   counter := 0 ;
   goto st1 ;
  }
 }
}
```

前の節で述べたように SFL において，ソフトウェアでのメソッドに相当するのは一連の state の遷移の定義，すなわち stage の定義であるので，OO-HDL での継承をメソッド単位で行おうとすると，わずか 1 つの state の追加や上書きの際にも stage 全体の再定義を必要とする．つまり，スーパークラス（継承する module）ですでに定義されている state を再利用することができない．このような state の追加や上書きは，継承によるハードウェア部品のカスタマイズにおいて，最も頻繁に行なわれることであると思われる．

継承によるハードウェア部品のカスタマイズにおいては，通常，その状態の追加や上書きが発生する．しかし，メソッドに相当するのは一連の状態遷移の定義（ステージの定義）であるので，継承をメソッド単位に行おうとすると，単なる状態の追加や上書きでもメソッド全体の再定義が必要となる．

この問題を解決する方法としては，並行ソフトウェアの場合と同様に記述を分離する，すなわち state およびその遷移を，stage の記述とは分離して記述する方策が考えられる．しかし，この分離記述はかえって全体を把握しにくくしてしまう可能性がある．この全体把握の困難性はソフトウェアでも問題となるところである．

ここで，OO-HDL は高級言語とは違い単純な構造であるため，非同期及び動的な状態遷移の変更を記述する必要が極めて少ないということが言える．従って，ソフトウェアで解決策として提案されている分離記述を用いたときの利点より，用いないことによる全体の把握の容易性の方が比重が高いと思われる．

そこで我々は最も簡明な方策として，メソッド単位だけでなく文単位の継承も可能にしてこれを解決することを検討している．これは，ハードウェア記述では状態遷移以外に文の間の依存関係がないことから，理論的にも破綻なく適用できるものと考えている．また，この文単位の継承は，SFL では各種の遷移や起動をユニークに指定するため，相性がいいと考えられる．この文単位の継承により，先の 2 つの問題点はいずれも完全に回避することができる．よって，次の言語要素を継承の対象として新たに加えることにする．

・state
・単位動作

なお，継承異常への対策は，VHDL のオブジェクト指向化案ではまだ全く検討されていない．また，IEEE 内部案の 1 つでは文単位の継承も検討されているが[8]，これは継承異常の回避を考えてのものではない．

5．オブジェクト指向ハードウェア記述言語 SFL++

5.1 オブジェクト指向機能

SFL の動作の記述はモジュール単位で，カプセル化されている．従って，オブジェクト指向化としては継承を導入することが中心となる．SFL++ は次節で紹介する拡張した機能も含め，基盤言語 SFL を拡張して作成しているため，言語の構文などはほとんど SFL と同じである．そして，SFL にクラスの概念を加え，サブモジュールをインスタンスとして取り扱う．クラスはパラメータを持ち，パラメトライズド・モジュールを可能にしている．

継承については，ここでは単純化のために単一継承とし，継承異常の回避のために文単位の継承も可能にしている．例として，先の継承異常が発生していた継承は単に下記のように記述するだけでよい．これは先に挙げた SFL の例を継承し，リセット機能を付け加えたものと同等な module である．スーパークラスは新しく加えた予約語“super”によって指定する．またこの例では文単位の継承も行なっており，“::”をつけることで，その内部への上書きや追加にスイッチされる．これによって，先ほど例示した継承異常を回避することが可能となる．

```
module counter_with_reset{
 super counter ;
 input reset ;
 stage::cnt{
  state_name st3 ;
  state::st1 par::{
   any::{ reset:goto st3 ; }
  }
  state st3 par{
   counter := 0 ;
   goto st1 ;
  }
 }
}
```

5.2 拡張的な機能

オブジェクト指向とは無関係であるが，オブジェクト指向の機能を十分に発揮させるために，SFL++ への配列の導入も併せて進めている．これは，モジュールや端子などの SFL の記述要素のほとんど全てについて，配列記述を可能にしようとするものである．パラメトライズド・モジュール定義は，この拡張の一つであるが，これによって，例えばパイプラインなど規則的なハードウェア構造の再帰的な定義なども可能にしようと試みている．

下記は，先に挙げたカウンタの例のビット幅をモジュールの引数とした例である．

```
module counter(bit) {
 circuit_type incre {
  input in<bit> ;
  output out<bit> ;
  instrin up ;
  instr_arg up(in) ;
  instruct up out = in + 0b1 ;
 }
 input in<bit>, flg ;
 output out<bit> ;
 instrin start ;
 reg counter<bit> ;
 incre inc ;
 stage_name cnt { task run(counter) ; }
 instruct start generate cnt.run(in) ;
 stage cnt {
  state_name st1,st2 ;
  first_state st1 ;
```

```
  state st1 par {
   counter := inc.up(counter).out ;
   out = counter ;
   any { flg : goto st2 ; }
  }
  state st2 par {
   counter := in ;
   goto st1 ;
  }
 }
}
```

当然ながら，このパラメトライズド・モジュールだけでは実際のモジュールは生成されない．例えば，先の例と全く同等なモジュールを生成したい場合には，

```
counter(10) count10 ;
```

として，モジュールのインスタンスを生成すればよい．これはサブモジュールタイプ名による，サブモジュールの生成と同等の記述である．

さらに下記ではパラメトライズ機能に加え配列機能も取り入れ，state を配列として記述した，より高度な例である．これは，カウンタを継承したもので，リセットを行なった後 N クロックだけ待ち，カウントを始めるカウンタである．

```
module counter_with_reset_wait(N){
 super counter ;
 stage::cnt{
 state st3[N] ;
 state::st1 par::{
  any::{ reset:goto st3[0] ; }
 }
 #for i = 0 #to N-1 {
  state st3[i] par{
   #if i = 0 #then counter := 0 ;
   #if i = N-1 #then goto st1 ;
   #else goto st3[i+1] ;}
  }
 }
}
```

5.3 処理系

先に挙げた機能拡張を実装するために，SFL++ を基盤言語 SFL に自動変換するプリプロセッサを作成した．既存の SFL ソースコードからも継承できるように，SFL，SFL++ のどちらも読み込めるように作成している．この処理系はプログラミング言語 Scheme で記述して作成した．

6．おわりに

本稿では，HDL のオブジェクト指向化として SFL を取り上げ，その継承異常の考察，および我々が設計開発している SFL++ の基本概念を述べた．

今後の課題として，文単位での上書きや追加を許す継承を提案したがその妥当性の理論的な検討を厳密に行なう必要がある．また，例えば並行ソフトウェアで問題となっている内部状態による受信メッセージの制御などは，ここでは考慮していないので，このようなより高いレベルでの継承機構についても探っていく．

加えて，SFL++ によって規則的ハードウェア構造の再帰的記述が可能になるため，例えば論理型や関数型言語による形式的な VLSI 構造記述から，直接的な書換えによって SFL++ プログラムを得て論理合成に持ち込むことができるようになるはずである．そこで，これまで我々が進めてきた VLSI アーキテクチャの形式的導出設計[9],[10]の研究をさらに押し進めて，アーキテクチャにとどまらず実際の VLSI 回路の導出まで実現したいと考えている．

参 考 文 献

(1) 安浦，山田（編），特集「ハードウェア記述言語」，情報処理，Vol. 33，No. 11（1992）

(2) S. Matsuoka, K. Taura and A. Yonezawa, "Highly Effcient and Encapsulated Re-use of Synchronization Code in Concurrent Object-Oriented Language", Proc. OOPSLA'93, pp. 109-126 (1993)

(3) L. Thomas, "Inheritance Anomaly in True Concurrent Object Oriented Languages : A Proposal", Proc. IEEE TENCON'94, pp. 541-545 (1994)

(4) 中村，小野，ULSI の効率的な設計法，オーム社（1994）

(5) http://www.kecl.ntt.jp/car/parthe/

(6) http://vhdl.org/vi/oovhdl/

(7) S. Swamy, A. Molin and B. Covnot, "OO-VHDL", IEEE Computer, Vol. 28, No. 10, pp. 18-26 (1995)

(8) M. Mills, "A Minor Syntax Change to VHDL Yields Major Object Oriented Benefits", IEEE DASC Object-Oriented VHDL SG (1995)

(9) N. Yoshida, "Transformational Derivation of Systolic Arrays", Concurrency : Theory, Language and Architecture (Ito and Yonezawa eds.), LNCS No. 491, Springer, pp. 297-311 (1991)

(10) 吉田, "関数型言語による抽象プロセス構造の高階記述およびその変換導出の試み", ソフトウェア工学の基礎 II (大蒔編), 近代科学社, pp. 131-139 (1996)