デバイスの記事

SystemCによるビヘイビア合成入門

第8回/最終回

ビヘイビア設計の流れと今後の動向

桜井 至

SystemC言語を入力とするビヘイビア合成技術の基礎について解説した連載の最終回をお届けする．今回は，アルゴリズム記述の作成からRTL記述の生成までの具体的な作業手順と各工程の注意点について説明する．生成されるRTL記述の性能にこだわる場合，ハードウェアの構成を意識しながらアルゴリズム記述に手を加えていく必要がある．（編集部）

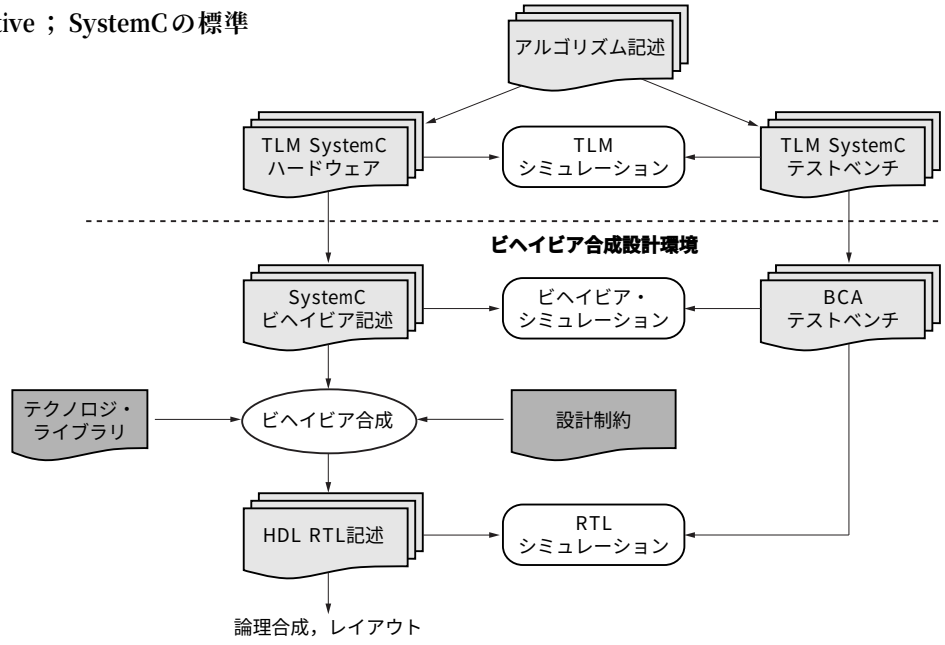
本連載も今回で最終回を迎えます．この講座では，ビヘイビア合成の基本的な機能や設計フロー，記述方法などについて紹介してきました．現在，実際の設計でビヘイビア合成ツールを利用している読者の方はまだ少ないと思いますが，ビヘイビア合成とは何かを知りたい方や，今後の導入を検討されている方の参考になったのであれば幸いです．

最終回になる今回は，アルゴリズム記述からハードウェア化を考慮したビヘイビア記述への変更の過程，今後の動向，OSCI（Open SystemC Initiative；SystemCの標準化・普及推進団体）の標準化作業などについて解説します．

● ビヘイビア・レベルの設計フローをおさらい

最初に，アルゴリズム記述を作成して実際にビヘイビア合成を実行するまでの設計フローについて説明します（**図1**）．

まず，アルゴリズム記述について，どの部分をハードウェア化するかの検討を行います．アルゴリズム記述の中にはハードウェア化に必要のない部分やソフトウェアとして実現する部分が含まれているので，それらを分離します．ハードウェア化する部分が決まったら，ハードウェア化を考慮した分割や記述の変更を行います．その際に，TLM（transaction level modeling）モデルによる協調検証環境を用いれば，システム検証やハードウェアとソフトウェアのトレードオフ解析が行いやすくなります．



BCA：bus cycle accurate
RTL：register transfer level
TLM：transaction level modeling

図1
ビヘイビア合成を利用した設計フロー
まず，アルゴリズム記述からハードウェア化する部分とそうでない部分を分ける．その際，TLMレベルの協調検証環境を用いれば，ハードウェアとソフトウェアのトレードオフ解析を行いやすい．次に，ハードウェア化する部分の構造を分割し，ビヘイビア合成可能な記述に変更する．ビヘイビア合成の結果を見ながら，設計制約の設定や記述の再修正を行い，目標とする性能を満足するRTL記述を生成する．

次に，ハードウェア化する部分の構造を分割します．分割にはモジュールやプロセスによる分割があります．ビヘイビア合成では，このような分割が最終的に得られるRTL回路の性能を決めるので重要な作業であると言えます．さらに，ビヘイビア合成できない記述をビヘイビア合成可能な記述に変更し，必要に応じてデータ・タイプのビット幅の指定を行います．

この時点で，ほぼビヘイビア合成が可能となりますが，合成の結果を見ながら，ビット幅の再指定や記述の変更，ループや配列の展開などの指示を行い，最終的に目標性能を満足するRTL（register transfer level）回路を得ます（p.143のコラム「ビヘイビア合成ツールの実例」を参照）．

なお，SystemCを用いてRTLと同等な記述レベルで設計することも可能ですが，この場合，従来どおりHDLで記述したほうが効率的で，検証も高速です．SystemCを使用するメリットはビヘイビア・レベルやTLMレベルで設計できることにあります．

以下では各工程について，順番に説明していきます．

● 性能重視の場合はハードウェア化を考慮して記述

ビヘイビア合成は，RTL記述より抽象度の高い記述を用いるため，その入力であるビヘイビア記述についてはハードウェア構造（アーキテクチャ）を考慮する必要はないと思われるかもしれません．C/C++を用いたアルゴリズム記述をそのままビヘイビア合成できる場合もありますが，目標とするハードウェアの性能を満足するRTL記述が生成される保証はありません．

必要な性能を備えるRTL記述を確実に得るためには，ある程度ハードウェア構造を考慮する必要があります．ハードウェア構造を考慮していないアルゴリズム記述を入力すると，期待はずれのレイテンシや面積を持つRTL記述が生成される可能性があります．そのため，一般には純粋なアルゴリズム記述を用いずに，ハードウェアを考慮したビヘイビア記述に書き換えます．

一方，従来のHDLによるRTL設計では，RTL設計を開始する時点で詳細なハードウェア構造を決める必要があります．レジスタの位置や必要な演算リソース，マルチプレクサ，動作を制御するステート・マシンなど，具体的なハードウェア構造に基づいてRTL記述を作成していきます．ビヘイビア合成を用いればこのような詳細な構造はツールが自動的に生成しますが，より上位の構造は設計者が考慮しなければなりません．

● 必要最低限のデータを扱うように書き換える

問題の理解を容易にするために，簡単なアルゴリズム記述の構造例を**リスト1**に示します．アルゴリズムを構成する各動作は関数として定義されています．関数の引き数にはデータ配列が渡され，計算された結果がデータ配列に戻ります．**リスト1**では，ライン長として4096の1次元配列を定義していますが，この配列サイズは，画像処理ではフレーム，画像ブロック，ライン長などになると思います．

アルゴリズム記述では，フローを組み立てやすいように引き数に配列をとり，それぞれが独立に動作する関数を作成することがあります．しかし，そのような関数で構成されたアルゴリズム記述をビヘイビア合成すると，配列はメモリにマッピングされます．その場合，それぞれの関数の間にメモリ・アクセスが入るため，処理に多くのサイクル数がかかることになります．また，配列をレジスタに展開するとレジスタ数が多くなるため，面積が増えてしまいます．いずれにしても，良いRTL記述が生成されません．

リスト1　簡単なアルゴリズム記述の構造例

```
#define MAX 4096

void algorism()
{

  int data[MAX];

  . . . . .

  // メイン・アルゴリズム
  // 関数A
  void func_A(int *data, ..) {
    int a;
    for(int i=0; i<MAX; i++) {
       a = *(data + i);
       // アルゴリズムA
          . . .
          *(data + i) = . . . ;
    }
  }

  // 関数B
  void func_B(int *data, ..) {
    int b;
    for(int i=0; i<MAX; i++) {
       b = *(data + i);
       // アルゴリズムB
       . . .
       *(data + i) = . . . ;
    }
  }

  // 関数B
  void func_C(int *data, ..) {
    int c;
    for(int i=0; i<MAX; i++) {
       c = *(data + i);
       // アルゴリズムC
       . . .
       *(data + i) = . . . ;
    }
  }

  . . .

}
```