RTL記述から動作記述への抽象化における組合せ回路の解析フロー

安楽 亮大郎[1,a)]　久我 守弘[1,b)]　伊藤 寛人[2,c)]　井戸 大介[2,d)]　飯田 全広[1,e)]

概要：過去に設計されたディジタル回路をIPとして再利用する際，RTL（Register Transfer Level）で開発された回路はサイクル精度で設計されているためタイミングを大きく変えるような再利用が困難である．一方で時間概念のない動作記述から高位合成によるアーキテクチャ探索を行うことで，様々な設計空間を短時間で設計することが可能となってきた．そこで，本研究ではRTLから時間概念のない動作記述に抽象化を行い，高位合成によるアーキテクチャ探索を行えるようにすることで，IPの再利用を容易にすることを目指している．本稿では，RTLの抽象構文木から得られるデータフローグラフの情報を基に，組み合わせ回路を例とした式抽出の処理フローについて報告する．

Analysis Flow for Combinational Logic from RTL to Behavior Abstraction

Ryotaro ANRAKU[1,a)]　Morihiro KUGA[1,b)]　Hiroto ITO[2,c)]　Daisuke IDO[2,d)]　Masahiro IIDA[1,e)]

Abstract: When reusing a digital circuit IP designed in the past, it is difficult to reuse the circuit developed by RTL because it is designed with the cycle accurate model. On the other hand, it has become possible to explorer widely design space by performing an architecture exploration by high-level synthesis from behavioral model. Therefore, in this research, we aim to facilitate the reuse of IP by abstracting from RTL to behavioral description and enabling the architecture exploration by high-level synthesis. We report the flow of formula extraction about the combinational circuit as an example, based on the data flow graph obtained from the abstract semantic tree of RTL.

1. はじめに

集積回路の集積度の向上に伴い，ディジタルシステムをSoC（System on a Chip）としてひとつの集積回路として実現できる時代においては，いかに大規模なシステムを短期間に設計・開発できるかが重要である．集積回路の設計開発手法は従来の机上でのゲートレベル設計に取って代わり，1990年代からはVerilog HDLやVHDL等のハードウェア記述言語を用いてレジスタトランスファレベル（RTL：Register Transfer Level）による設計が主流となった．近年ではRTLよりもさらに抽象度の高いビヘビアレベル（Behavior Level）を用い，C言語ライクな記述から高位合成（High-Level Synthesis）により回路設計・開発を行えるようになってきた．より抽象度の高いレベルで設計できることから，設計期間や検証期間の短縮を図ることができるようになった．一方，従来のRTLやゲートレベルで開発されてきた設計資産は多くの集積回路設計関連企業等において蓄積されている．これらの設計資産をIP（Intellectual property）として再利用することも，集積回路の集積度向上に伴う設計期間や検証期間の増大に対処できる方法として有用である．

しかし，RTLやゲートレベルによる設計資産は再利用が難しいという問題点がある．ゲートレベルの論理回路は，実装デバイスに依存したセルライブラリを用いて最適化さ

[1] 熊本大学, Chuo-ku, Kumamoto-shi 860–8555, Japan
[2] 三菱電機エンジニアリング, Higashi-ku, Nagoya-shi 461–0048, Japan
[a)] anraku@st.cs.kumamto-u.ac.jp
[b)] kuga@cs.kumamoto-u.ac.jp
[c)] Ito.Hiroto@ma.mee.co.jp
[d)] Ido.Daisuke@ma.mee.co.jp
[e)] iida@cs.kumamoto-u.ac.jp

れているため，他の実装デバイス向けにはそのまま利用することが難しい．また，ゲートレベルよりも抽象度の高い RTL 記述の論理回路においても，その動作はクロックに同期し決められた状態遷移に従って動作するサイクル精度レベル（Cycle Accurate Level）で実現されているため，再利用の際に状態遷移や機能ユニット数等をターゲットシステムに合わせて最適化することが困難である．

一方，近年実用化されてきた高位合成を用いると，時間概念のない Untimed Functional レベルで記述されたコードから，アーキテクチャ探索機能により演算器等の機能ユニット数や状態遷移数が異なる様々な論理回路を自動生成することが可能であり，開発するアプリケーションに適応した論理回路の設計空間を広げることができるメリットがある．

そこで，本研究では RTL で開発された過去の設計資産を一度 Untimed Functional モデルの動作記述へと抽象化し，高位合成を利用することで設計資産の広範囲な再利用を可能にする手法について研究を行う．具体的には，高位合成が行っている処理を逆に辿るアプローチであり，サイクル精度レベルで記述された RTL から CDFG（Control Data Flow Graph）を生成し，データパスと状態遷移に着目して Untimed Functional モデルの動作記述を抽出する方法である．本稿では，この RTL から動作記述への抽象化について，組合せ回路をターゲットとした逆変換フローの一事例について報告する．

以下，2 章では逆変換処理の関連研究について 2.1 節で示した後に，2.2 節において我々が提案する処理フローの概要について述べる．続く 3 節では組合せ回路をターゲットとして，変換事例を示しながら抽象化処理の詳細について述べる．4 章において，変換前のコードと逆変換後のコードの等価性について議論する．最後に 5 章を本稿のまとめとする．

2. RTL からの逆変換技術

2.1 関連研究

RTL 記述から C/C++/SystemC コード等へ逆変換を行うツールは既にいくつか存在しており，有名なものとして Verilator[1] や v2c[2] などがある．一般にサイクル精度で記述されている RTL では機能検証のための動的シミュレーションに時間がかかる．そのため，これらのツールはいずれも RTL をほぼ同じ記述レベルのまま C コードへ変換し，機能レベルでの動的シミュレーションをプロセッサのネイティブコードで行えるようにすることを目的としている．したがって，C コードへ変換する際により抽象度の高い動作記述である untimed functional モデルまで変換したり，高位合成可能な C コードまで変換することを目的としていない．

東芝の特許 [3] も同様に RTL を C コードへ変換し高速なシミュレーションを行うことを目的としている．なお，この特許では RTL の状態遷移における各状態において，データフローを状態数分だけ再構築して作成する手法が取られている．そのため，各状態において動作しない冗長な記述を削除しており，一部抽象化を施しているといえる．しかしながら，時間概念は未だ残る timed functional モデルまでの変換となっている．

高位合成におけるアーキテクチャ探索をにより設計空間を広げることを目的として，高位合成可能な C コードまで抽象化することを目指した研究に Veriintel2C [4] がある．これは香港理工大学の Department of Electronic and Information Engineering で開発された RTL から高位合成可能な C コードまで逆変換するツールである．VeriIntel2C では，RTL の動作を表すためにペトリネットによりモデリングを行い，その情報から CDFG（Control DataFlow Graph）を作成する．そして得られた CDFG から C コードを生成する．なお，C コードへの抽象化の際にはループや配列構造の識別が必要となる．そのために，RTL より得られたデータフローグラフからグラフマッチングにより，ループ化および配列化可能なグラフ構造を部分グラフとして検出することにより，ループ・配列構造を再構築する方法を採用している．

Veriintel2C は RTL 記述による設計資産の再利用を目的としたものであるため，我々の研究目的に近く本研究を進める上で参考になる．なお，我々の研究と VeriIntel2C との差異として，ペトリネットを使用せず RTL の構文解析から得られる抽象構文木（AST：Abstract Semantic Tree）より直接 CDFG を抽出する．また，VeriIntel2C は構文解析器として商用パーサである Verific Parser Platform を用いて作成されているが，我々はオープンソースで提供されているツール群を利用することで，広く利用できるツールとしての開発を目指している．

2.2 提案処理フローの概要

我々が提案する逆変換ツールの処理フローを図 1 に示す．一般的な言語処理システムと同様に，Verilog HDL により記述された RTL に対して構文解析を行い抽象構文木（AST：Abstract Semantic Tree) を作成する必要がある．構文解析にはオープンソースの Verilog HDL 用構文解析器である Pyverilog[5] を用いた．Pyverilog はパーサとして PLY（Python Lex-Yacc）を用いており Python のみで処理できる．Pyverilog 内の Verilog HDL 構文解析器は vparser であり，これにより得られた AST からコントロールフローおよびデータフローを容易に取り扱うことが可能である．そのため，Verilog HDL の言語処理を行うために有用なツールであると判断した．

逆変換ツールの開発にあたり，以下の方針で開発を行う．

RTL 記述から逆変換により時間概念のない動作記述へ抽

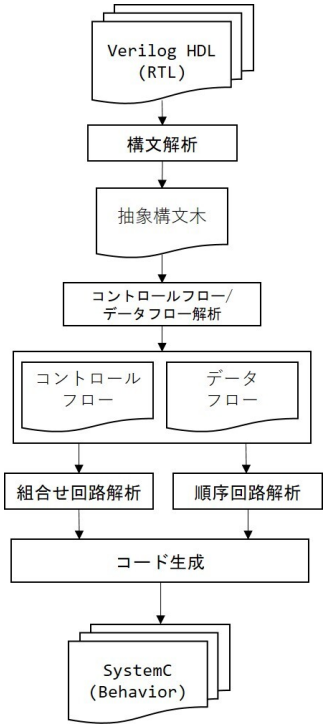

図 1　逆変換の処理フロー

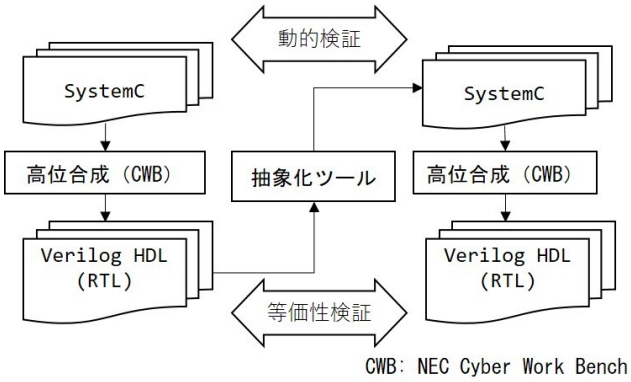

図 2　逆変換ツールの開発フロー

象化する際には，先ず論理回路を組合せ回路と順序回路を分けて考えることにする．RTL 記述での組合せ回路は，数式あるいは論理式として記述されている．そのため組合せ回路については，AST からデータフローグラフを得ることができれば，数式あるいは論理式のレベルでの変換は容易であるといえる．しかし，順序回路については，論理合成により生成されるフリップフロップ（FF：Flip-Flop）が状態遷移を実現するステートマシンを構成するものであるのか，あるいは，データパス上で一時的にデータを記憶しておくためのレジスタであるのかを分けて考える必要がある．本稿では，先ず最初の取り組みとして，組合せ回路に限定して逆変換を行うこととする．

また，RTL で記述されたコードは基本的に CDFG を意識した記述になっているが，人手で記述されたコードは様々な方針により記述されている可能性があり，逆変換のためのルールを抽出するのに手間がかかると考えられる．そのため，本研究では図 2 に示すように SystemC で記述されたコードを NEC 社製の高位合成ツールである Cyber Work Bench（CWB）[6], [7] により RTL 化し，その RTL コードを入力として逆変換を行うことを最初のステップとした．入力を CWB が生成した RTL コードに限定することで，RTL に含まれる CDFG 情報以外にも，変数名等の情報を利用して逆変換を行うことができる．もちろん，本ツールを汎用的に利用できるようにするためには，先々 CWB が生成した RTL だけでなく，ハンドコーディングされた RTL コードに対しても適用できるように拡張していく予定である．

逆変換によって得られた SystemC コードが正しく変換できているかについては，SystemC コードのレベルでシミュレーションによる動的検証により確認を行う．また，静的検証による等価性チェックについても行う．逆変換により得られた SystemC コードを再度 CWB により高位合成を行い，得られた Verilog HDL の RTL コードを逆変換前の RTL コードと等価性チェックを行うことで，逆変換により得られたコードが等価であるか否かを判断することとした．

3. 処理フローの詳細

3.1 サンプル回路

前章で述べたように Verilog HDL コードを Pyverilog の vparser を用いて構文解析い，得られた AST を利用して組合せ回路部分の数式・論理式抽出を行う．変換フローを説明するために，2 つの組合せ回路を例として取り上げる．ひとつは選択信号を S とする 2 入力 1 出力のマルチプレクサ（MUX1），他方はリセット信号によるゲーティング回路（MUX2）である．それぞれ，図 3 および図 6 に示す SystemC コードを用意し，高位合成ツール CWB を用いて RTL へ変換する．変換後の RTL コードはそれぞれ図 4 および図 7 となった．この RTL を vparser で処理し，得られた AST から手作業で作成した CDFG を，それぞれ図 5 および図 8 に示す．

3.2 AST のデータ表現

RTL において組合せ回路は assign 文や always 文により表現されている．構文解析器 vparser により得られる AST において，組合せ回路に関する部分は IfS 回路であれば図 5，IfR 回路であれば図 8 のように木構造で表現されている．それぞれのツリー構造の分岐点（ノード）には演算子のデータタイプが葉の部分にはオペランドのデータタイプが割り当てられている． そのデータタイプの内部に演算子や変数の情報が含まれているためこの情報をもとに数式・論理式に変換する．

```
SC_MODULE(MUX1){
  sc_in<bool> S;
  sc_in<bool > a,b;
  sc_out<bool > y;

  SC_CTOR(MUX1)  {
    SC_METHOD(method0);
  }

  void method0() {
    if (S){
        y=a;
    } else {
        y=b;
    }
  }
};
```

図 3 IfS の動作記述（SystemC）

```
module MUX1 ( S ,a ,b ,y );
input  S;
input  a;
input  b;
output y;
reg    y ;
reg    y_c1 ;

always @ ( b or a or S )
  begin
    y_c1 = ~S ;
    y = ( ( { 1{ S } } & a )
      | ( { 1{ y_c1 } } & b )
      );
  end

endmodule
```

図 4 高位合成後の IfS RTL（Verilog HDL）

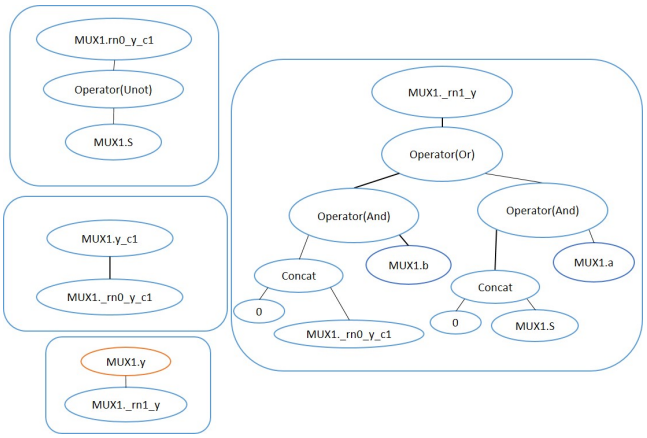

図 5 IfS RTL の CDFG

3.3 式への変換

AST 内に含まれるデータタイプから回路の最終出力となる変数を抽出する．そして，その最終出力と初期入力変数以外の共通の変数がある際は，AST を辿ることで検索し接続関係を得る．これにより最終出力から初期入力の変数まで辿ることができ，式としてまとめることが可能となる．

式抽出の際に特に注意すべき点として，ハードウェア記述言語に特有のビット操作（partselect）がある．また，if 文による分岐の取扱いについても注意が必要である．以下，この 2 つの処理の方法についてまとめる．

（1）ビット抽出

AST に含まれるノードの，データタイプが `DFpartselect` である場合は，ビット抽出情報を表現している．ビット抽出はハードウェア記述言語にみられる特有の記述であり，例えば `A[4:2]` の様な記述は複数ビットから成る変数 A の第 4 ビットから第 2 ビットの 3 ビットを抽出することを意

```
SC_MODULE(MUX2){
  sc_in_clk    CLK;
  sc_in<bool>  RESET;
  sc_in<bool>  a;
  sc_out<bool> y;

  SC_CTOR(MUX2)  {
    SC_METHOD(method0);
      sensitive << RESET.pos();
      sensitive << CLK.pos();
  }

  void method0() {
    if (!RESET){
        y=a ;
    } else {
        y=0;
    }
  }
};
```

図 6 IfR の動作記述（SystemC）

```
module MUX2 ( CLK ,RESET ,a ,y );
input         CLK;
input         RESET;
input   [7:0] a;
output  [7:0] y;
reg     [7:0] y_r;

assign  y = y_r ;
always @(posedge CLK or posedge RESET)
  if ( RESET )
    y_r <= 8'h00;
  else
    y_r <= a ;

endmodule
```

図 7 高位合成後の IfR RTL（Verilog HDL）

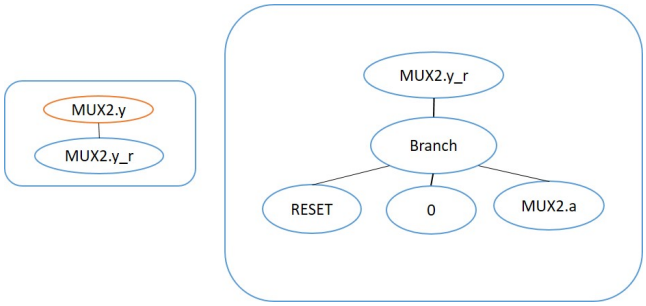

図 8 IfR RTL の CDFG

味する．ノードには最上位ビットと最下位ビットの情報が含まれるが，最上位ビットと最下位ビットが同じであれば 1 ビットの抽出となり，同じでない場合はそのまま配列の最上位ビットと最下位ビットに割り当てられる．

（2）分岐の取り扱い

AST に含まれるノードのデータタイプが DFBranch である場合は，if 文による条件分岐を表現している．DFBranch の情報を含む場合は逆変換行う際にそのまま if 文の形でコード生成することができる．しかし，図 3 の SystemC コードをを CWB により合成した図 4 の RTL コードは，if 文を DFBranch としてではなく論理式による表現へ最適化している．この場合，論理式部分を事前に評価し True 部と False 部をそれぞれ復元しなければ，元の if 文へ逆変換することができない．逆変換による式の抽象化を目指す場合，このような処理を適宜実施する必要がある．

4. 検証および考察

4.1 検証

前節で述べた抽象化方法を用いて逆変換処理を行う．入力は RTL で記述された Verilog HDL コードであるが，現状では CWB による高位合成によって生成された Verilog HDL コードのみを対象としている．出力となるコードは SystemC を対象とする．SystemC を選択した理由のひとつは，ハードウェア記述に依存している記述を SystemC では素直に表現できるからである．なお，得られる式表現をさらに抽象化する場合でも，C ライクな言語であれば容易に表現できることも理由として挙げられる．図 9 および図 10 に逆変換により SystemC へコード生成した結果をそれぞれ示す．

```
SC_MODULE(MUX1_E){
  sc_in<bool> S, a, b;
  sc_out<bool> y;
  sc_uint<8> _rn0_y_c1 ;

  void p_MUX1_E(void){
    _rn0_y_c1=(~S);
    y=((S&a)|(_rn0_y_c1&b));
  }

  SC_CTOR(MUX1_E){
   SC_METHOD(p_MUX1_E);
  }
};
```

図 9 逆変換後の IfS の動作記述

```
SC_MODULE(MUX2_E){
  sc_in_clk    CLK;
  sc_in<bool>  RESET, a;
  sc_out<bool> y;

  void p_MUX2_E(void){
    if(RESET){
      y=0;
    }else{
      y=a;
    }
  }

  SC_CTOR(MUX2_E){
    SC_METHOD(p_MUX2_E);
    sensitive << CLK.pos();
    sensitive << RESET.pos();
  }
};
```

図 10 逆変換後の IfR の動作記述

逆変換後に得られる動作記述 SystemC コードが，元の SystemC コードと同じであるかについて検証を行った．検証方法として以下の (1) 〜 (3) 方法により実施した．動的シミュレーションによる検証では CWB 上において SystemC のシミュレーション機能を使用した．また，等価性検証は Synopsys 社の formality により 2 つの Verilg HDL コード間の等価性チェックを行った．

(1) 目視および机上でのコード確認
(2) 動的シュミレーションによる検証（動的検証）
(3) 等価性検証（静的検証）

先ず，目視では，変換前の SystemC コードと逆変換後に得られた SystemC コードの比較において，IfR 回路ではほぼ同等であったが，IfS 回路ではかなり変化していた．IfR 回路では，図 6 と図 10 を比較してみると，条件判定が反転して取り扱われており，意味的には等価であることが分かる．一方，IfS 回路では，図 3 と図 9 を比較してみると if 文がなくなり入力 S を条件として論理和を用いて出力が判断されている式が生成されていることが分かる．式の SystemC の記述で行数も異なり，if 文が記述されていない変換結果になっているが，机上で式の評価を行ったところ入出力や計算結果に違いはないと判断した．実際，IfR 回路，IfS 回路ともに，動的シミュレーションを行った結果，シミュレーション結果は返還前の SystemC コードと逆変換後の SystemC 記述は，それぞれ同じシミュレーション結果をを示しており，逆変換は正しく実行されていると判断した．

さらに，逆変換後の 2 つの動作記述を，それぞれ CWB により再高位合成を行い，RTL 記述での等価性検証を行った．その結果，IfS 回路，IfR 回路ともに，Verilog HDL の記述は異なっていたが，等価性検証では同じものと判断された．

4.2 考察

これらの検証結果から，if 文により記述された 2 入力 1 出力マルチプレクサやゲーティング回路程度の組合せ回路であれば式抽出が可能となった．しかしながら，現状ではまだ制約事項が多い．例えば，フリップフロップを含み，クロックイベントの前後で変数の値がフィードバックするようなループ含む回路の場合，そのような構造であるこ

との検出と対応が必要であり，現ツールでの逆変換は対応できていない．また，組合わせ回路だけでなく状態遷移を含む順序回路の対応も必要である．さらに，抽象化できるVerilog コードとしては CWB で高位合成したものにのみ対応しているため，ハンドコーディングされた RTL への対応も検討する必要がある．今後はより汎用性のある逆変換ツールへの拡張を図りたい．

5. おわりに

本稿では，RTL で開発された過去の設計資産を動作記述へと抽象化し高位合成を利用することで再利用を可能にする手法について述べた．本稿によって組合わせ回路に対する分岐の取り扱いを提案することで動作記述での if 文を含む回路が対応可能となった．今後は作成した抽象化ツールに対してループや順序回路に対しても対応できるようにして汎用性をあるツールの完成を目指したい．

謝辞 本研究を進めるに当たり，研究室 OB である新玲於奈氏には多大なるご協力をいただいた．ここに感謝致します．

参考文献

[1] W. Snyder, P. Wasson and D. Galbi : Verilator, https://www.veripool.org/projects/verilator/wiki/Intro, 2017.
[2] Rajdeep Mukherjee, Michael Tautschnig and Daniel Kroening : v2c – A Verilog to C Translator Tool, http://www.cprover.org/hardware/v2c/.
[3] 秋葉剛史，五十嵐真悟：ハードウェア動作記述変換方法及びそのためのプログラム，特許 4153732，株式会社東芝，2008/9/24.
[4] Anushree Mahapatra and Benjamin Carrion Schafer : “VeriIntel2C: Abstracting RTL to C to maximize High-Level Synthesis Design Space Exploration,” Integration, the VLSI Journal 64 (2019) pp.1–12, Elsevier, 2019.
[5] 高前田（山崎）伸也 : Pyverilog, https://github.com/PyHDI/Pyverilog/
[6] NEC Corporation : CyberWorkBench, https://jpn.nec.com/cyberworkbench/index.html
[7] 若林一敏：“ソフトウェアプログラムからハードウェア記述を合成する高位合成技術—プロセッサ以外の汎用プログラム実行機構—，” IEICE Fundamentals Review Vol．6，No．1，pp.37-50，2012/7.
[8] Benjamin Carrion Schafer,Zi Wang : High-Level Synthesis Design Space Exploration: Past, Present, and Future https://ieeexplore.ieee.org/abstract/document/8847448
[9] Anushree Mahapatra , Benjamin Carrion Schafer : Optimizing RTL to C Abstraction Methodologies to Improve HLS Design Space Exploration https://ieeexplore.ieee.org/abstract/document/8702355
[10] Nicola Bombieri , Franco Fummi , Graziano PravadelliAbstraction of RTL IPs into embedded software https://dl.acm.org/doi/pdf/10.1145/1837274.1837283