RTL から動作レベルへの抽象化における配列・ループ処理

桑原尚大[1]　久我守弘[2]　飯田全広[3]

概要：従来の RTL（Register Transfer Level）やゲートレベルで開発されてきた多くの設計資産は，集積回路設計関連企業等において蓄積されている．一般に RTL は論理回路の状態遷移等が設計時の回路に最適化されており，他の設計への再利用が非常に困難である．そこで，RTL 記述を抽象化する際の配列・ループを含む動作記述へ抽象度を高める逆変換を行い，近年実用的に利用できるようになってきた高位合成技術を用いて再設計できるようにする．再設計時に高位合成のアーキテクチャ探索により設計時の選択肢を広げることができるため非常に有用である．本研究では，これまで研究グループで開発してきた抽象化技術ベースとして，構文解析の結果から配列やループに関する記述情報を抽出し配列・ループ処理に関するマッチングや新たな中間表現を追加することで，配列・ループを含む高位合成向けの動作レベル記述を生成する手法を提案する．検証結果から，現状では，制限はあるものの RTL 記述を配列・ループ処理を含む形で抽象化することが可能となった．

キーワード：高位合成，抽象化，IP 再利用

1. はじめに

従来の RTL（Register Transfer Level）やゲートレベルで開発されてきた多くの設計資産は，集積回路設計関連企業等において蓄積されている．一般に RTL は論理回路の状態遷移等が設計特定の回路に最適化されており，他の回路への再利用が非常に困難である． 一方で，近年では RTL よりもさらに抽象度の高い動作レベル（Behavioral Level）を用い，C 言語ライクな記述から高位合成（High-Level Synthesis）による回路設計・開発が実用的に行えるようになってきた．高位合成により抽象度の高いレベルから設計でき，またアーキテクチャ探索機能を利用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．

そこで，本研究では 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設計資産の広範囲に再利用可能にすることを目指す．本研究は本研究グループの先行論文においてコントロールフローを含む RTL 記述を対象として，転送表を用いた抽象化手法を提示した．研究グループの既存の研究ではデータフローのみの RTL 記述を対象とした抽象化手法を提示し中間変数の取り扱いを新たに追加していた．さらに，先行論文[2]ではパターンマッチングによるノードの変換や，コード生成時の記述順序の解析を追加することにより幅広い RTL 記述を抽象化する手法を提示した．

本稿では，これらの先行論文で対応できなかった RTL 記述を抽象化する際の配列・ループを含む動作記述を生成することで再利用時の高位合成においてアーキテクチャ探索の幅を広げることを目的としていた．先行研究の手法を踏襲し，構文解析の結果から配列やループに関する記述情報のマッチングや新たな中間表現を追加することで，配列・ループを含む HLS 向けの動作レベル記述を出力する手法を提案する．

2. 先行研究

2.1 関連研究

RTL 記述から C++コード等の記述に変換するツールは既に存在しており，Verilator[3]や v2c[4]などがある．しかし，これらのツールはいずれも RTL を C コードでシミュレーションすることを目的としているため，動作記述レベルまで抽象化するものや，高位合成可能な C コードまで変換するものではない．東芝の特許[5]も同様に RTL の C 記述でのシミュレーションを目的としたものである．しかし，この特許では，各状態においてデータフローを状態数分だけ再構築作成する手法がとられている．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されているといえる．そして，最近では高位合成可能な C まで戻すことのできる.v2C の後継である Veriintel2C[6]が開発されている．東芝の特許と Veriintel2C は本研究の目的を達成するために大いに参考となる．

2.2 研究グループにおけるこれまでの研究動向

本研究は研究グループの既存の研究と 2 つの先行研究の後継研究である．先行研究は，安楽の行った先行論文[1]と高木の行った先行論文[2]がある．

安楽による研究では，データフローのみの RTL 記述を対象とした抽象化手法を提示した．中間変数の取り扱いによりデータフローを連結し，最終的に数式として出力する手法を提示した．ビット選択，パート選択を読み込める機能を追加する等，新の研究と比較すると読み込める RTL 記述の幅が広がった．

高木による研究では，コード生成時の記述順序の解析やパターンマッチングによるノードの変換を追加することに

1 熊本大学 大学院自然科学教育部
2 熊本大学 大学院先端科学研究部
3 熊本大学 半導体・デジタル研究教育機構

より, 安楽の研究に加え幅広い RTL 記述を抽象化する手法を提示した.

3. 提案手法

3.1 提案手法概要

RTL から動作レベル記述への抽象化変換における配列・ループ処理を生成する利点は以下のとおりである.

(1) 可読性・拡張性の向上

- 配列やループが展開された場合と比べると,構造がコンパクトで直感的に把握でき理解しやすい.
- 再利用の際に記述の変更や拡張が容易になり, 変更後にコード全体が冗長化するのを防ぐ.

(2) HLS（高位合成）での最適化が容易

- アーキテクチャ探索の幅が広がることで,ループ内部のパイプライン化や並列化が適用しやすくなる.
- シミュレーション速度が向上し,実行時間の短縮に繋がる.

本研究における処理の流れは, 先行研究をベースとして, 配列・ループ化に対応する機能を追加することにより実現することを目指した. 開発するツールにおける処理手順の概要を以下に示す. また, 本ツールの処理フローを図 1 に示す.

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[7]を用いた. 入力とする VerilogHDL コードに対して, Pyverilog を用いることにより解析情報から抽象構文木（Abstract Syntax Tree, AST）を生成し解析することで for ループの有無を確認, for ループが存在する場合はそのまま配列・ループ処理を行うために必要な情報を中間表現（Intermediate Representation, IR）として構築する. for ループを含まない場合は, 数式処理から配列・ループの構築が可能な処理を特定・解析し, ループ処理を行うために必要な情報を中間表現（Intermediate Representation, IR）として構築する. その後 Jinja2 テンプレートを用いたコード生成により配列・ループを含む SystemC を出力する.

1. Verilog コードの解析(AST 生成)
2. IR（中間変数）への変換
   - AST から数式の配列構造を解析.
   - for ループの有無を確認
     i. ループが存在する場合
        - for ループの解析・最適化.
     ii. ループが存在しない場合
        - 計算パターンやインデックスの一貫性から行列ベクトル積を認識.
        - 行列ベクトル積であれば for ループを構築.
        - マッチングに失敗すると代入式の連続として出力.
3. SystemC コードの生成

本ツールのフローを図 1 に示す.

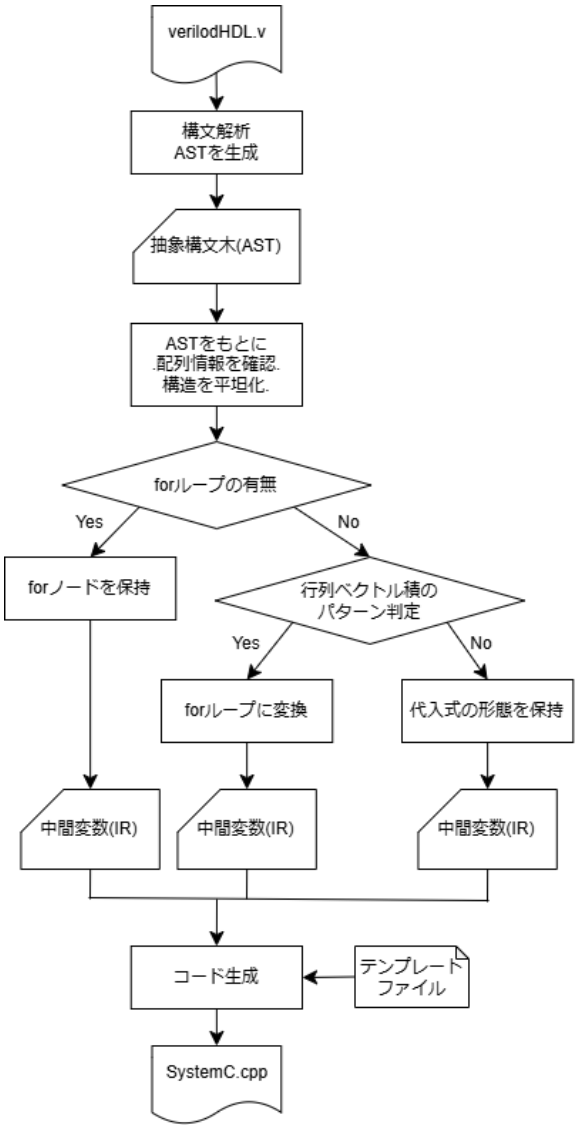

図 1：提案するツールのフロー図
Fig.1：Flow of the proposed tool.

3.2 提案手法の詳細

図 2, 図 3 に示した for ループが存在する場合と展開されている場合それぞれの行列ベクトル積のサンプルコードを例に, for ループを含む Verilog コードと, 行列ベクトル積が展開された Verilog コードの両方から配列･ループ処理を含んだ SystemC を生成するまでの処理全体についてそれぞれ説明する.

```
module sample_4x2(
    input [7:0] A[3:0][1:0],  // 4x2 matrix
    input [7:0] B[1:0],       // 2-dimensional vector
    output [15:0] C[3:0]      // Output 4-dimensional vector
);
    integer i, j;
    always @(*) begin
        for (i = 0; i < 4; i = i + 1) begin
            C[i] = 0;
            for (j = 0; j < 2; j = j + 1) begin
                C[i] = C[i] + A[i][j] * B[j];
            end
        end
    end
endmodule
```

図 2：for ループを含む Verilog コード
Fig.2：Verilog sample code with for loop.

```
module sample_4x2_unrolled(
    input [7:0] A[3:0][1:0],
    input [7:0] B[1:0],
    output reg [15:0] C[3:0]
);
    always @(*) begin
        // Row 0
        C[0] = (A[0][0] * B[0]) + (A[0][1] * B[1]);
        // Row 1
        C[1] = (A[1][0] * B[0]) + (A[1][1] * B[1]);
        // Row 2
        C[2] = (A[2][0] * B[0]) + (A[2][1] * B[1]);
        // Row 3
        C[3] = (A[3][0] * B[0]) + (A[3][1] * B[1]);
    end
endmodule
```

図 3：for ループが展開された Verilog コード
Fig.3：Verilog code with for loop unrolled.

3.2.1 構文解析

Pyverilog を用いた Verilog コードの解析により AST を生成する．ここで得られる AST は Python における変数表現で記憶される．図 4 に数式表現（右辺の一部）を例として示す．

```
right=Plus(
    Plus(
        Times(
            Pointer(Pointer(Identifier('A'), IntConst('0')), IntConst('0')),
            Pointer(Identifier('B'), IntConst('0'))
        ),
        Times(
            Pointer(Pointer(Identifier('A'), IntConst('0')), IntConst('1')),
            Pointer(Identifier('B'), IntConst('1'))
        )
    ), …
```

図 4：生成される AST の例
Fig.4：Example of generated AST.

3.2.2 中間表現(IR)変換

次に，解析された AST を中間表現(IR)に変換する．まず AST から数式の配列構造を確認し，その後に for 文の有無で処理を分岐する．

1. AST から数式の配列構造を確認.
   i. AST 内のノードを再帰的に走査.
   ii. 配列・数式の形式に文字列化.

この時点での数式(の一部)は下図 5 のようになる．

```
"type":"Always",
"data":[
    {"type":"Subst","left":"C[0]",
    "right":"( (A[0][0]*B[0]) + (A[0][1]*B[1]) + (A[0][2]*B[2]) + (A[0][3]*B[3]) )"},
    {"type":"Subst","left":"C[1]",…},
    …
    {"type":"Subst","left":"C[5]",…}    ]
```

図 5：文字列化された数式
Fig.5：stringified formula.

2. for ループの有無を確認.
   i. AST 内 always ブロック内で ForStatement ノードを探索.
      A) always ブロック内に ForStatement ノードが存在すれば，for ループとして認識.
   ii. for ループを含む場合.
      A) for ループの解析
      B) for ループは，IR 内で条件式(i<4)・ステップ(i++)・ボディ(C[i]=A[i]+B)の 3 つの主要な部分に分けて扱う.
      C) IR への変換

この時に生成される IR は下図 6 のように for ループの構築に必要な情報がまとめられている．

```
"operations":[
  {
   "type":"For","pre":"i=0","cond":"i<4","post":"i++",
   "body":[
    {"type":"Subst","left":"C[i]","right":"0"},
    {
     "type":"For","pre":"j=0","cond":"j<2","post":"j++",
     "body":[
      {"type":"Subst","left":"C[i]",
       "right":"C[i] + (A[i][j]*B[j])"}
     ]
    }…
```

図 6：for ループを含む.Verilog コードの IR 例
Fig.6：IR example for Verilog code with loops.

3. for ループを含まない場合
   ① ループ再構成の可能性を特定
   数式が for ループによりループ再構できるかについて特定する．行列ベクトル積の例を示す.
      i. 数式処理部を列挙.
         A) 代入式(Subst)を取り出してリスト化.
         B) リスト化した代入式の左右の辺に対しマッチングを行う.

リスト化された代入式は図 7 のようになる．

ii. 左辺の検出．
- A) 変数名の統一性を確認．
- B) 添え字の連続性．

iii. 右辺の検出．
- A) 右辺の文字列を解析．
  各項が A[N][M]*B[M]を繰り返し足し合わせる形になるかをマッチング．
- B) 各項毎の解析．
  - 各項の構造から添え字を読み取る．
  - 左辺の添え字の一致・繰り返し回数への該当・連続性を確認．

これらのマッチング処理を経て，行列ベクトル積を確定・ループ処理を施した IR を生成．

② 行列ベクトル積ではない場合は代入式の連続とした IR を作成．

ここで生成される中間変数(IR)の例を下図 8 に示す．

```
[
    {"type":"Subst","left":"C[0]","right":"(A[0][0]*B[0])+…+(A[0][3]*B[3])"},
    {"type":"Subst","left":"C[1]",…},
    …
    {"type":"Subst","left":"C[5]",…}
]
```

図 7：リスト化された代入式
Fig.7：Listed assignment expression.

```
{
 "module_name":"sample_6x4_unrolled",
 "ports":[(省略)],
 "operations":[
  {
   "type":"For","pre":"i=0","cond":"i<6","post":"i++",
   "body":[
    {"type":"Subst","left":"C[i]","right":"0"},
    {
     "type":"For","pre":"j=0","cond":"j<4","post":"j++",
     "body":[
      {"type":"Subst","left":"C[i]",
       "right":"C[i] + (A[i][j]*B[j])"}
     ]
    }…
```

図 8：展開された 6x4 行列ベクトル積の IR
Fig.8：IR of expanded 6x4 matrix-vector product.

3.2.3 SystemC コード生成

コード生成において生成するコードはハードウェア記述言語である SystemC 形式を採用した．SystemC による動作記述は，大枠としてメソッドとその実行の構造がある．動作記述は，変数宣言部と処理部からなる．これらの構造を満たすようにコード生成を行う．IR を解析することで，SystemC の SC_METHOD を用いたコードを生成する．

処理の流れは以下の通りである．

1. IR を解析
2. ポート宣言部分
   IR の ports 情報を生成．(例: sc_in<sc_int<width>> A[. . .])
3. 処理の本体部分
   IR の operations(for，Subst，Assign などのノード)を再帰的に変換し for(…){…}や, .write()文などを出力．
4. システムのクロック管理
5. SystemC コード全体の生成
   Jinja2 テンプレート・固定フォーマットを用いて，SC_MODULE(module_name)の枠組みに上記 2，3，4 を挿入．

4. 評価

今回提案するツールの評価を行う．

4.1 実装環境

本研究で作成したツールを実装した環境は以下である．

- OS : Windows 11pro
- Python version 3.11
- エディタ，デバッグ支援ツール ：.Visual Studio Code
- 主要なライブラリ
  - PyVerilog version 1.3.0
  - Jinja2 version 3.1.2

4.2 検証

今回提案するツールに関して, 想定した Verilog コードを入力し出力されたコードに対し抽象化ができているか，ループの構築ができているかを検証する．

4.2.1 検証方法

検証の方法として, 動作レベルと RTL 記述の等価性検証を行うツールが研究室になかったため，研究室内の以前の研究においては作成した逆変換ツールにより生成した SystemC コードについて高位合成を用いて得られる RTL 記述と, 逆変換ツールによる抽象化変換前の RTL 同士の等価性検証で確認していた．本研究ではその前段階として目視での検証結果を示す．

4.2.2 検証対象

検証対象としていくつかの行列ベクトル積を行う VerilogHDL 記述を生成した．検証対象として用いる VerilogHDL コードを表 1 に示す．

表 1：検証対象
Table.1：Verification target

種類	名前(.v)	特徴
1.	Sample_4x2	4x2 行列ベクトル積を計算する
2.	Sample_8x8	8x8 行列ベクトル積を計算する(サイズ違い)
3.	Sample_4x2_renamed	変数名が異なる 4x2 行列ベクトル積
4.	Sample_4x2_unrolled	展開された 4x2 行列ベクトル積
5.	Sample_6x4_unrolled	展開された 6x4 行列ベクトル積(サイズ違い)
6.	Sample_4x2_ur_rn	変数名が異なる展開された 4x2 行列ベクトル積
7.	Sample_1x4_no_for	ループを持たないスカラ積の代入文の連続

4.2.3 検証結果

検証対象としていくつかの行列ベクトル積を行う VerilogHDL 記述を生成した．検証対象として用いる VerilogHDL コードを表 2 に示す．

表 2：検証結果
Table.2：Verification results

種類	名前(.sc)	抽象化	ループ処理
1.	Sample_4x2	○	○
2.	Sample_8x8	○	○
3.	Sample_4x2_renamed	○	○
4.	Sample_4x2_unrolled	○	○
5.	Sample_6x4_unrolled	○	○
6.	Sample_4x2_ur_rn	○	○
7.	Sample_1x4_no_for	○	○

for ループを含む行列ベクトル積に関して，抽象化前の sample_4x2.v と抽象化後の sample_4x2.sc をそれぞれ図 9，図 10 に示す．

for ループを含む Verilog コードに関して，抽象化変換後に生成されたコードから，SystemC コードへの抽象化とコード内の配列・ループ処理の構築を確認できる．

また Sample_8x8 の検証結果から，行列ベクトル積のサイズによらず，行列ベクトル積 NxM に対応していることが確認できた．また Sample_4x2_renamed の結果から，変数名にも依存していないことが確認できた．

```
module sample_4x2(
    input [7:0] A[3:0][1:0],  // 4x2 matrix
    input [7:0] B[1:0],       // 2-dimensional vector
    output [15:0] C[3:0]      // Output 4-dimensional vector
);
    integer i, j;
    always @(*) begin
        for (i = 0; i < 4; i = i + 1) begin
            C[i] = 0;
            for (j = 0; j < 2; j = j + 1) begin
                C[i] = C[i] + A[i][j] * B[j];
            end
        end
    end
endmodule
```

図 9：抽象化前の Sample_4x2.v
Fig.9：Sample_4x2.v before abstraction.

```
#include <systemc.h>

SC_MODULE(sample_4x2) {
  sc_in<bool> clk;


  void process() {

    for (int i = 0; i < 4; i++) {

      C[i].write(0);

      for (int j = 0; j < 2; j++) {

        C[i].write((C[i].read() + (A[i][j].read() * B[j].read())));

      }

    }

  }

  SC_CTOR(sample_4x2) {
    SC_METHOD(process);
    sensitive << clk.pos(); // always@(*) => simplified
  }
};
```

図 10：抽象化後の Sample_4x2.sc
Fig.10：Sample_4x2.sc after abstraction.

また，展開された行列ベクトル積に関して，抽象化前の sample_4x2_unrolled.v と抽象化後の sample_4x2_unrolled.sc をそれぞれ図 11，図 12 に示す．

for ループが展開されている Verilog コードに関しても，抽象化変換後に生成されたコードから for ループを構築可能な数式処理を特定し，SystemC コードへの抽象化とコード内の配列・ループ処理の構築の実現を確認できる．

Sample_6x4_unrolled の検証結果から，展開された for ループにおいて配列や行列のサイズに依存しない抽象化とループの構築がともに実現できたと言える．

また Sample_4x2_ur_nn の結果からは，こちらの場合においても変数名の依存がないことを確認できた．

Sample_1x4_no_for では行列ベクトル積以外での抽象化，ループの構築を実現できたことで，今後の分岐追加による機能の向上が可能であると示すことができた．

```
module sample_4x2_unrolled(
    input [7:0] A[3:0][1:0],
    input [7:0] B[1:0],
    output reg [15:0] C[3:0]
);
    always @(*) begin
        // Row 0
        C[0] = (A[0][0] * B[0]) + (A[0][1] * B[1]);
        // Row 1
        C[1] = (A[1][0] * B[0]) + (A[1][1] * B[1]);
        // Row 2
        C[2] = (A[2][0] * B[0]) + (A[2][1] * B[1]);
        // Row 3
        C[3] = (A[3][0] * B[0]) + (A[3][1] * B[1]);
    end
endmodule
```

図 11：抽象化前の Sample_4x2_unrolled.v
Fig.11：Sample_4x2_unrolled.v before abstraction.

```
#include <systemc.h>

SC_MODULE(sample_4x2_unrolled) {
  sc_in<bool> clk;


  void process() {

    for (int i = 0; i < 4; i++) {

      C[i].write(0);

      for (int j = 0; j < 2; j++) {

        C[i].write(C[i].read() + (A[i][j].read() * B[j].read()));

      }

    }

  }

  SC_CTOR(sample_4x2_unrolled) {
    SC_METHOD(process);
    sensitive << clk.pos(); // always@(*) => simplified
  }
};
```

図 12：抽象化後の Sample_4x2_unrolled.sc
Fig.12：Sample_4x2_unrolled.sc after abstraction.

4.3 考察

検証結果より，制約条件はあるものの RTL 記述の抽象化変換後に配列・ループ処理を構築することに成功した．今回想定した範囲では，RTL 内に for ループが存在する場合と展開されている場合それぞれに関して，行列のサイズや変数名への依存なく抽象化変換後のループの構築が可能であった．加えてループを持たない代入の連続の処理に関しても抽象化変換後にループ処理を構築することができた．

現在のツールは，特に行列ベクトル積やスカラ積のような計算処理に焦点を絞っている．具体的に対応可能な Verilog コードの範囲は以下の通りである．

I. 基本的な代入操作（assign や always 内の代入文）
assign 文や always ブロック内での基本的な代入式（C[i] = A[i] + B[i]など）を解析し、IR を通じて SystemC コードへと変換可能.

II. 配列や行列操作
多次元配列（A[4][2]）やベクトル（B[2]）に対するアクセスをサポートしており，特に行列とベクトルの積を含む操作（A[i][j] * B[j]）が適切に変換される．

III. for ループの扱い
for ループを解析し，インデックス変数の範囲設定やループ内での計算（例えば，行列の掛け算や累積加算）に対応.

IV. 簡単な算術演算
ツールは加算、乗算、累積加算といった基本的な算術演算を解析して、IR の形式に変換が可能である．

また，本研究を通してツールの正当性を目指すうえでたくさんの課題も見つかった．以下に改善すべきと考えられる点を示す．

(1) 抽象化変換後のコードの正当性評価が現在目視である点：
前述の通り，現状 RTL 記述の抽象化により生成した動作記述が，元の RTL 記述と一致しているかを静的に確認する方法はなく，また現状では抽象化の正確さを測る指標もないため，今後の課題としては出力された SystemC を実際に高位合成にかけ，正常にコンパイルが可能であるか，合成後の RTL と入力に用いた RTL との整合性などを調査するべきである．

(2) 研究室における既存の抽象化ツールとの連携が不完全である点：
Pyverilog を解析の軸にしている点や，入出力にそれぞれ VerilogHDL, SystemC を用いている点で共通しているため連結自体のコストは高くないと思われるが，連携後は抽象化ツールの入力制限による開発の不自由が予想されるため，現在は未連携である．

(3) ツールの汎用性について：
本研究で提案したツールは想定した数式処理や記述によるコードに特化した構造になっており，入力する Verilog 次第では全く対応ができなくなってしまう．そのため対応するパターンを増やすことでコードへの対応を増やすことが可能になる．今後の課題として，構文解析で得た情報から記述によらず同一の処理を特定する手法を示したい．

(4) エラーハンドリングの強化：
現在のエラーメッセージはある程度の指針を示してくれますが、ユーザーが提供する Verilog コードのエラー（例えば、構文エラーや未対応の構造）に対する詳細なエラーハンドリング機能を強化することも改善すべき点である．

5. まとめ

本論文では，VerilogHDL の記述を配列・ループ処理を含む高位合成可能な動作記述へと変換する手法を提案した．構文解析の結果から for ループの有無で分岐し，ループがある場合は for ループを特定し，中間表現を経て配列・ループ処理を含む動作記述（SystemC）を生成する．ツールを検証したところ，制約は多いが一部の記述に対し配列・ループ処理を含む抽象化を行うことに成功した．

そのため，過去の資産を再利用するという最終的な研究目的のうち，配列・ループ処理を行う点では達成可能性を示せたといえる．ただし，現状では非常に制約が多いので，制約事項のうち優先度の高いものから順に解決していく必要がある．実装を進めてより汎用性のある逆変換ツールを作成することが今後の研究の展望である．

謝辞 本研究を進めるに当たり，多大な御助言，御鞭撻を頂いた熊本大学大学院自然科学教育部情報電気電子工学専攻尼崎太樹教授に深く感謝いたします．そして研究にあたって，貴重なご助言を頂いた研究室の諸先輩である新玲央奈氏，安楽遼太郎氏，高木彬氏，ならびに苦楽を共にした同輩の皆さんに感謝致します．

参考文献

[1] 安楽 遼太郎, 久我 守弘, 伊藤 寛人, 井戸 大介, 飯田 全広：" 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," 火の国情報シンポジウム 2021，情報処理学会九州支部，March. 2021.RTL
[2] 高木 彬，久我 守弘，飯田 全広，伊藤 寛人，井戸 大介："パイプライン動作する演算モジュールの RTL 記述から動作記述への抽象化," 火の国情報シンポジウム 2022，情報処理学会九州支部，March. 2022.
[3] W. Snyder, P. Wasson, D. Galbi: Verilator. https://www.veripool.org/projects/.verilator/wiki/Intro, 2017
[4] Rajdeep Mukherjee, Michael Tautschnig, Daniel Kroening: “v2c-A Verilog to C Translator Tool.” http://www.cpro.ver.org/hardware/.v2c/
[5] 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” 特開 2004-02184，株式会社東芝，2004 年 1 月 22 日
[6] Anushree Mahapatra, Benjamin Carrion Schafer,”VeriIntel2C:Abstracting RTL to C to maximize High-Level Synthesis Design Space Exploration Integration”, the .VLSI Journal 64, 1/12, 2019
[7] 高前田（山崎）伸也: ”PyVerilog: A Python-based Hardware Design Processing Toolkit.” https://github.com/PyHDI/Py.Verilog
[8] David C. Black, Jack Donovan, et al.: “SystemC: From the Ground Up” 2nd Edition, Springer, 2010
[9] Philippe Coussy, Adam Morawiec: “High-Level Synthesis: From Algorithm to Digital Circuit,” Springer,2008