C-026

上位ハードウェア設計言語 Melasy+ による NuSMV コード生成と設計検証

Design, Verification and NuSMV Code Generation based on A Meta Hardware Description Language : Melasy+

西田 翔 †　魚住 有記歌 ††　和崎 克己 †††
Sho Nishida　Yukika Uozumi　Katsumi Wasaki

1　はじめに

IT 技術の発展により様々な環境で電子機器が動作している．社会システムの多くがこれらの技術と信頼性に依存しており，社会システムの維持には信頼性の高いハードウェアが不可欠である．ハードウェアの信頼性を維持するためには，組み込み自己テスト等の機能を備える必要があり，回路規模が増大・複雑化する．ハードウェア設計においては，高い信頼性を確保し，効率良く設計・検証できる環境が求められている．高級言語で対象システムを実装し，コード生成によって対象システム向けのオブジェクトや実行可能モジュールを得るコンパイラが開発された．近年，ハードウェアの設計を記述するために比較的高度な言語によって書かれたコードから，直接，回路の構成情報を生成する，ハードウェアコンパイラが使われている．

設計の正当性を形式的にチェックするツールとしては，NuSMV[1] 等のモデル検査 [2] ツールが存在する．これらのツールを用いることによって設計の正当性を自動で評価することができる．しかし，NuSMV を用いてモデル検査を行うためには極めて低級な言語を利用する必要があり，VHDL[3] 等の言語で設計したシステムを，検証目的のために改めて NuSMV で記述する必要がある．同じハードウェアに対して異なる言語で複数回記述することは，設計の一貫性を保つことが困難であり，また工程管理の面からも非効率的である．これらの問題を解決するためには，記述性に優れ，設計から開発・実装までを一貫して行うことができる開発環境が必要である．

我々の研究グループでは，ハードウェア記述言語，モデル検査用の言語など，様々な既存言語向けコードを自動生成し，検証から実装までを一貫して行う目的で上位ハードウェア記述言語 Melasy+[4] の開発を行なってきた．Melasy+ は上位設計記述から XML 中間表現系を経て，記述に対する構造解析 [5] や各種コード生成を行う．従来の Melasy+ においては，中間表現生成器とコード生成器は独立していた．今回，対象言語の拡張が容易になるようコード生成器の再設計を行い，中間表現系とコード生成系の統合を図った．再設計した環境を用いたケーススタディとして，自己修復機能付き FIFO 型メモリ [6] と静止画像向けコーデック [7] の上位設計記述と，記述した FIFO 型メモリに対する NuSMV モデル検査器による検証を行った．

† 信州大学大学院工学系研究科，Graduate School of Science and Technology，Shinshu University.
†† 株式会社スターインフォテック，Star Info Tech Co.,Ltd.
††† 信州大学工学部，Faculty of Engineering，Shinshu University.

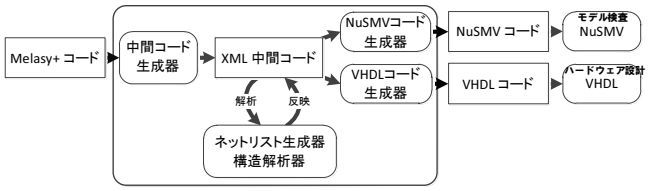

図 1　Melasy+ コンパイラ処理系の位置づけ

2　上位ハードウェア記述言語 Melasy+

2.1　概要

Melasy+ は NuSMV や VHDL など，既存の処理系の上位に位置する処理系である．Melasy+ は，中間コード生成器と各種処理系向けコード生成器から構成される (図 1)．中間コード生成器は C++ のクラスライブラリとして実装されており，Melasy+ コードは C++ の言語機能を用いて記述を行う．Melasy+ コードを中間コード生成器に与えることにより，XML 形式の中間表現コードを得る．得られた中間表現コードを各種処理系向けコード生成器に与えることにより，各言語のコードを得る．

2.2　言語仕様

Melasy+ コードは C++ のクラス拡張として記述し，文法や機能は C++ を利用する．ただし，Melasy+ の記述において値を保持するために，int や char 等の C++ 標準の型は利用できない．代わりに，Melasy+ では値の保持に Logic 型や Digit 型を用いる．Logic 型は 1 ビットの値を保持し，Low と High の 2 状態を表す型である．Digit 型は固定長ビットの値を表す型であり，テンプレート引数にビット幅を指定することができる．値の代入には int 型の値，あるいは const char 型の文字列を用いる．また，Logic や Digit は配列として宣言することが可能である．Logic または Digit の配列は，入出力の指定は変数名のみで行うことができるが，動作の指定はそれぞれ配列の添字を用いて個別に行う必要がある．

Melasy+ は上位設計を記述する際，部分回路をコンポーネント単位で記述し，コンポーネントの連なりや入れ子構造でハードウェア全体を構成する．コンポーネントの記述は，Component クラスが持つ機能を利用して行う．入出力は in，out，sync などの関数を用いて定義する．出力の定義には switch_，case_，default_ などを

用いて特殊な条件分岐構文を利用可能である．switch_式は評価する変数を引数に一つと，それに続く [] 演算子の引数に，複数の条件と出力の対を受け取る．条件と出力の対は case_ によって記述し，いずれの場合にも当てはまらなかった場合の出力を記述するために，default_を使用することが可能である．C/C++ の switch 文と同等の動作を行うが，case 中に一つの式しか持つことができない点で異なる．これは，C/C++ の switch 文が処理の流れを制御する為の構文であるのに対して，Melasy+ の switch_が回路の出力の一つを制御するための式である．定義されたコンポーネントは関数として呼び出し，インスタンス化して利用する．回路記述内の再利用可能な箇所は，同一定義を呼び出すことでコードの再利用が可能である．

2.3 繰り返し構造の記述

規則的な構造を簡潔に記述するために，C++ の持つ制御構文を利用することができる．for 文を用いることで，同様の動作を複数回指定する際に記述が容易になる．また，if 文による条件分岐で繰り返し処理の動作を制御することが可能である．規則的な構造を繰り返すような部分回路に対して，規則的な構造の先頭と末尾にのみ例外的な接続情報を与え，他の部分回路を均質に扱うようなことが可能である．さらに，C++ 特有の機能であるテンプレートを Melasy+ の記述に利用することが可能である．構造上にビット幅などを持つ部分回路の場合，テンプレート機能を用いてビット幅を与えることができる．テンプレートを適用する部分回路は，ビット幅が可変長であることを前提とした記述を行うことで，スケーラブルな記述を行うことができる．

2.4 XML 中関系

Melasy+ は各種処理系に一つのコードを対応させることを目的としている．そのため，様々な処理系向けコード生成器に対応しやすいよう，XML 形式を用いた中間表現コードを経由する．XML 形式の中間表現コードを得るには，中間コード生成器である C++ のクラスライブラリを include した Melasy+ コードを，C++ コンパイラで出力した生成器を実行することで XML 形式の中間表現を得る．XML 中間表現コードは，上位記述内に配列やテンプレートなどの機能を用いて抽象的に表現された記述は全て展開され，具体的な値とラベルの解決が行われており，回路構造の把握に必要な情報を全て保持している．

2.5 下位言語向けコード生成器

Melasy+ では，中間コード生成器で生成した XML 形式の中間表現ファイルを，コード生成器に与えることでコード生成を行うことが可能である．現在，NuSMV と VHDL の二種類がコード生成に対応している．従来の Melasy+ 環境では，コード生成器は Melasy+ の他の機能とは別の言語で作成されており，独立していた．そのため，中間表現コード生成後，各言語向けコードを再度生成する必要がある．また，拡張性が乏しく，異なる言語向けコード生成器の新規対応が容易でない．この問題点を解決するため，コード生成器を C++ のクラスライブラリとして再設計を行った．

3 Melasy+ を用いた回路記述の上位設計例と構造的検査

3.1 構造的検査手法

Melasy+ では，生成した中間表現コードより，ソフトウェア上に回路構造を再現するネットリストを再生成することが可能である．再生成したネットリストを探査することにより，Melasy+ コードの表す回路構造に対して構造解析を行うことができる．構造解析の機能として (1) 回路の階層構造解析レポート (2) レジスターレジスタ間のクリティカルパス提示 (3) 非同期ループ構造の検出，ならびに (4) 回路のみ使用箇所検出，などが実行可能である．ネットリスト解析によって，回路の構成上の誤りや冗長箇所を検出することができる．また，静的構造解析で得られた回路に関するメタデータを中間表現コードへと反映することも可能である．

3.2 記述例 (1) 自己修復機能付き FIFO メモリ

記述例として，自己回復機能付き FIFO 型メモリ [6] を Melasy+ を用いて記述した．図 2 に FIFO 型メモリを構成する機能ブロック図を示す．DFF，Cell，Flag でメモリの一段分のユニットを構成し，直列につなぎ合わせることによりメモリを構成する．DFF はデータが格納される実態を表す．Cell は各ユニットの基本的動作を司るオートマトンを表す．待ち行列中の有効データは Flag が持つフラグ値で表す．自己修復機能付き FIFO 型メモリは，フラグに一時的な障害が発生した際に，有限回数のメモリ操作を行うことにより正常な状態に回復する機能を持つ．FIFO 型メモリを Melasy+ によって記述したコードの一部を図 3 に示す．Melasy+ の言語機能である for 文を利用して，Cell の繰り返し構造に対する接続情報を簡潔に記述し，テンプレート機能を用いてスケーラブルな設計記述を実現している．

3.3 記述例 (2) 静止画像向けコーデック

記述例として，静止画像向けコーデックである 2 進桁圧縮方式 (以下 BPC)[7] のデコーダを，Melasy+ を用いて記述した．BPC とは，0/1 のビット列で与えられるデータを，0　または 1 の値 (以下 Value) がいくつ連続しているか (以下 Length) という情報で捉え，Value と Length の組み情報を符号化テーブルを基に圧縮するエンコード方式である．BPC デコーダは，圧縮されたデータを符号化テーブルと照らし合わせて復号する．BPC デコーダのステートマシンを図 4 に示す．

デコーダは復号の逐次リアルタイム性を確保するために，符号の展開が完全に終わる前に展開データの出力を開始する．コードの展開とは，Value と Length を明らかにすることである．このうち，Value はコードの先頭 bit に示されている．よって，先頭 bit に示されている Value の出力をステートマシンを動作させるクロックに合わせて開始し，それと並行してコードを読みながら Length を明らかにしている．この場合，決定に要したステップ数が決定された Length の値以下である必要がある．このため，Length が小さい場合の展開処理は，圧縮データの読み込みにおいて 3bit シフトを用いた先読みを行い，短いステップ数で処理が完了する．

r_0，State，d_n，w はステートマシンの各状態，矢印付近の 3bit の数字 (Q_{10}，Q_9，Q_8) は遷移条件で，圧縮されたデータから読み込んだビット列の先頭 3bit を表

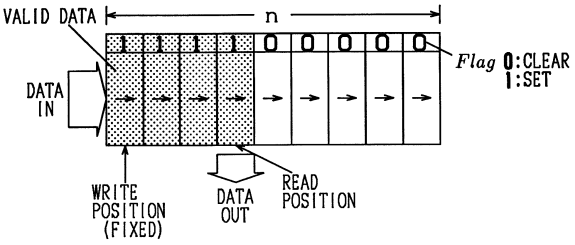

図 2　FIFO 型メモリ機能ブロック図

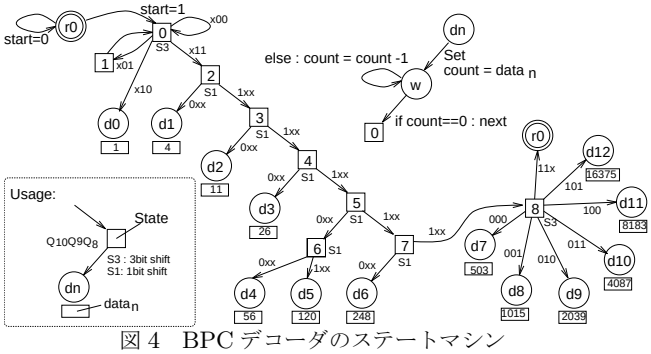

図 4　BPC デコーダのステートマシン

```
class Cell : public Component{
public :
  Logic read,own,next_f,clear,gate,top,ack;
  Digit<4> state;

  Cell()
  { in(read); in(own); in(next_f); out(clear);
    out(gate); out(state); out(top); out(ack);
…}

template<int N>
class Cyg_main : public Component{
public :
  Logic clock,ack_sum,fsum;
  CPU cpu; Cell cell[N]; flag flag[N];
…
  Cyg_main(){
…
   for(int i=0;i < N;i++){        //接続情報
    if(i < N - 1){
     PortMap pmc[] = {
    cell[i].read <= cpu.read,
    cell[i].own  <= flag[i].own,
    cell[i].next_f <= flag[i + 1].own };
     instance(cell[i],pmc);
    }else{
     PortMap pmc[] = {
    cell[i].read <= cpu.read,
    cell[i].own  <= flag[i].own,
    cell[i].next_f <= '0' };
   instance(cell[i],pmc); } }
…
```

図 3　Melasy+ によって記述した FIFO 型メモリ (抜粋)

```
class BPC_Decoder : public Component{
public :
  Logic clock,start,q8,q9,q10,shiftFlag,
       shiftNum,count0;
  Digit<14> countNum;
  Logic outputValue;     //出力中の値
  Digit<5> state;      //状態

  BPC_Decoder(){
…
    sync(state,
      switch_(state)[
        case_("00000",
          switch_(q9)[
            case_('0',
              switch_(q8)[
                case_('0',state),
                case_('1',"00001")]),
            case_('1',
              switch_(q8)[
                case_('0',"10000"),
                case_('1',"00010")])]),
        case_("00001","00000"),
        case_("00010",
          switch_(q10)[
            case_('0',"10001"),
            case_('1',"00011")]),
        case_("00011",
          switch_(q10)[
            case_('0',"10010"),
            case_('1',"00100")] ),
…
```

図 5　Melasy+ によって記述した BPC デコーダ (抜粋)

す．状態 d_n に対応する $data_n$ は，その次の状態へ遷移を行うとき，カウンタを用いて同一の値を n 回出力することを表す．State の下に書かれた記号が，S1 の場合は次の状態に遷移する際に読み込んでいる圧縮データを 1bit シフトし，S3 の場合は 3bit シフトすることを示す．BPC デコーダのステートマシンを，Melasy+ を用いて記述したコードの一部を図 5 に示す．

4　コード生成とモデル検査

4.1　モデル検査

モデル検査とは，開発対象であるシステムや回路が，求められる仕様を満たすものであるのかを，設計段階において検査する技術である．NuSMV 等のモデル検査ツールによって検査を行うことが可能である．モデル検査では，検査対象をモデル検査ツールの専用言語で，オートマトンと対応付けられた状態機械として記述し，モデルの満たすべき仕様を時相論理を用いて記述する．検査はモデルの取りうる全ての状態に対して網羅的に行われる．設計段階で開発対象が仕様を満たすのか検査することで，後に不具合が発見される可能性が減少する．開発段階の下流工程で判明する不具合を減らすことで，その対処にかかる開発コストを削減することができる．

4.2　NuSMV コード生成

NuSMV コード生成においては，XML 中間表現コードで表現された検査対象を，NuSMV の言語機能で表現し直す必要がある．NuSMV では，module 宣言，変数宣言部，遷移宣言部によって検査対象を記述する．module 宣言では，コンポーネント名と入力定義を与える．XML 中間表現コードよりコンポーネント名とポート情報を取得し，NuSMV コードの生成を行なっている．変数宣言部では，インスタンス定義におけるインスタンス名と接続情報，出力定義におけるポート名が記述される．出力定義においてはデータ型の表現方法の違いから，Melasy+ 言語での Logic 型を NuSMV における boolean 型に，Digit<N>型を word[N] 型に変換する．遷移宣言部は出力定義毎に宣言され，init と next の 2 つの関数によって，ポート名とその値の推移を記述する．XML 中間表現コードより出力定義の型，ポート名，接続情報を元に遷移記述を生成する．

```
SPEC A[CPU_0.write=1 U (flag.own = 1 & flag_1.
    own = 1 & flag_2.own = 1) -> AG(
    Collecter_1.o = 0)]
```

図 6　書き込み処理による修復を表す CTL 式

```
SPEC AG( (CPU_0.read=1 & (Cell_0.ack = 1 |
    Cell_1.ack = 1 | Cell_2.ack = 1)) -> AG(
    Collecter_1.o = 0) )
```

図 7　読み出し処理による修復を表す CTL 式

4.3　テストケース 自己修復機能付き FIFO 型メモリ

ケーススタディとして，Melasy+ で記述した FIFO 型メモリの上位設計 (前述 3.2) より，NuSMV コード生成とモデル検査を行った．前述した Melasy+ による FIFO 型メモリの記述より生成したコードと，CTL によって記述された検査式を用いて，モデル検査ツール上で回路モデルの振る舞いが，仕様を満足することを確認する．

FIFO 型メモリは書き込み処理が発生した場合，データと有効データフラグがシフトする．FIFO 型メモリは有効データのフラグを管理する Flag に一時的な障害が発生した場合，書き込み処理を繰り返すことにより，故障した Flag は有限ステップ内にフラグからから消えるため，少なくとも，メモリの段数分の書き込み処理によって正常な状態が保証される．書き込み処理による修復を表す CTL 式を図 6 に示す．

また，FIFO 型メモリは読み出し要求に対して，各 Cell がデータ待ち行列の先頭と判断すれば，他に先頭であると判断した Cell の有無に関わらず，読み出し処理を行う．これにより，読み出し処理によって Flag がクリアされるため，最悪のケースでもメモリ段数分の読み出し処理によって正常な状態となる．読み出し処理による修復を表す CTL 式を図 7 に示す．

図 8 に FIFO 型メモリの NuSMV コードを示す．図 8 中の flag_1 が，エラーとして [flag_1=1] で故意に指示してある．このように初期故障を意図的に埋め込んだ NuSMV コードと検査式をモデル検査上で駆動する．[flag_1=1] だけでなく，他に考えられる状況も同様の検査を行った結果，FIFO 型メモリの自己修復機能が仕様通りに働くことが確認できた．

5　まとめ

Melasy+ のコード生成器を再設計した．コード生成器を再設計することで，他の言語対応への拡張性を高め，既存の他機能との親和性を高めた．また，ケーススタディとして具体的に上位設計からコード生成，モデル検査まで行い，Melasy+ 環境としての一貫した流れを確認した．今後の課題として，上位記述としての抽象度を高め，機能レベルでの繰り返し構造記述などを実現したい．

謝辞　本研究は，梨和恭平氏 (2011 年度本学情報工学科卒) によって開発されたコード生成器を用いている．本研究の一部は科学研究費（23500174）の助成を受けたものである．

```
MODULE main
VAR
  CPU_0 : CPU(((Cell_0.ack|Cell_1.ack)|Cell_2.
      ack),clock,flag_sum_0.sum);
  Cell_0 : Cell(flag_1.own,flag_0.own,CPU_0.
      read);
  Cell_1 : Cell(flag_2.own,flag_1.own,CPU_0.
      read);
  Cell_2 : Cell(0,flag_2.own,CPU_0.read);
  Checker_0 : Checker(flag_0.own,1);
  Checker_1 : Checker(flag_1.own,flag_0.own);
  Checker_2 : Checker(flag_2.own,flag_1.own);
  CollecterDigit_3_0 : CollecterDigit_3((((
      Data_buff_0.out_put |
  Data_buff_0.out_put) |Data_buff_1.out_put) |
  Data_buff_2.out_put));
  Collecter1 : Collecter((((Checker_0.error |
      Checker_0.error) |
  Checker_1.error) | Checker_2.error));
  Data_buff_0 : Data_buff(CPU_0.data,Cell_0.
      gate,CPU_0.write);
  Data_buff_1 : Data_buff(Data_buff_0.d,Cell_1.
      gate,CPU_0.write);
  Data_buff_2 : Data_buff(Data_buff_1.d,Cell_2.
      gate,CPU_0.write);
  flag_0 : flag(Cell_0.clear,0,1,CPU_0.write);
  flag_1 : flag(Cell_1.clear,1,flag_0.own,CPU_0
      .write);
  flag_2 : flag(Cell_2.clear,0,flag_1.own,CPU_0
      .write);
  flag_sum_0 : flag_sum(((flag_0.own | flag_1.
      own) | fag_2.own));
  clock : boolean;
ASSIGN
  init(clock) := 0;
  next(clock) := case
  clock=1 : 0;
  clock-0 : 1;
esac;

MODULE Cell(next_f,own,read)
VAR
  ack : boolean;
  clear : boolean;
  gate : boolean;
  state : word[4];
  top : boolean;
ASSIGN
  init(ack) := 0;
  next(ack) := case
  state=0b4_0100 : case
  read=0 : 0;
  read=1 : case
  top=1 : 0;
  top=0 : 1;
esac;
```

図 8　FIFO 型メモリの NuSMV コード (抜粋)

参考文献

[1] NuSMV : a new symbolic model checker, http://nusmv.irst.itc.it/.
[2] E.M.Clarke, O.Grumberg, D.Peled : “Model Checking” ; MIT Press, 2000.
[3] VHDL : VHSIC Hardware Description Language, http://vhdl.org/.
[4] 白鳥 航亮, 和崎 克己 : “上位ハードウェア設計言語 Melasy+ による VHDL コード生成と動作検証” ; FIT2010（第 9 回情報科学技術フォーラム）講演論文集, 1, (C-002), 371-374, 2010.
[5] 西田 翔, 和崎 克己 : “上位ハードウェア記述言語 Melasy+ が生成した中間表現に対するネットリスト生成と静的解析” ; 平成 23 年度電気関係学会東海支部連合大会講演論文集, (I1-1), 2011.
[6] 和崎 克己, 不破 泰, 江口 正義, 中村 八束 : “セルオートマトンの概念を用いた自己回復能力をもつ通信用バッファ” ; 電子情報通信学会論文誌, Vol.J77-D-I, No.1, pp.41—52, 1994.
[7] 和崎 克己, 不破 泰, 江口 正義, 中村 八束 : “画像処理に適した高速リアルタイム復号が可能な 2 値画像符号とその評価” ; 画像電子学会誌 Vol.25, No.6, pp734-742, 1997.