A-031

モデル検査系に対応する上位ハードウェア設計言語 Melasy

A Meta Hardware Description Language “Melasy” for Model Checking system

岩崎 直木† 和崎 克己†
Naoki Iwasaki and Katsumi Wasaki

1. まえがき

IT 技術の発展により，様々な環境で電子機器が動作しており，年々規模・数ともに増加傾向にある．また，様々な社会システムがこれらの技術と信頼性に依存しており，とりわけディジタルシステムにおける技術と信頼性は社会システムの維持に必要不可欠である．仕様通りに確実に動作する信頼性の高いシステムを，より安価に効率よく設計できる環境が望まれている．

高級言語で対象システムを実装し[1][2][3][4]、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ための、コンパイラとして，VHDL[5]，Verilog，SystemC[6]等が開発された．

ハードウェア設計の正当性を形式的にチェックするツールとしては，SMV[7][8], NuSMV[9] 等のモデル検査ツール(Model Checker)が存在する．これらのツールを用いることで設計の正当性の評価を自動で行うことができる．しかし，NuSMV によって実ハードウェアの設計を全て記述するには，極めて低級な言語を利用しなければならず，VHDL や Verilog 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 等の言語で記述しなおす工程が新たに発生する．また，同じ設計を異なる言語で複数回行うことは，実際の開発現場の状況に鑑みるとコスト・納期などの制約において困難である．

本研究の目標は，既存のハードウェア記述言語，モデル検査用の言語，ならびにシミュレーションや協調設計向けの SystemC 向けのコード生成器を有するような、上位ハードウェア記述言語 "Melasy" の設計と，コンパイラの開発である．このシステムでは，コンパイル時に与えるコンパイルオプションにより，既存の様々な処理系向けコードの出力を可能にする．これにより，Melasy コードのみで，設計の検証から実機の設計まで行うことができる．

これまでに、関数型プログラミング言語 Haskell[10]とパーサライブラリ Parsec[11]を使用し，NuSMV 向けのコードを生成することに成功した．本コンパイラ系を用いてシステムを記述し，様々な言語向けのオブジェクトコードを生成することで，従来の手法で困難さが生じていた問題が解決できる範囲，あるいは，上位言語 Melasy による設計では，どのような構文上の利点や記述性が優れているのか，について簡単なケーススタディによって比較・評価する．

2. 関連研究

2.1 HDCaml

HDCaml[12]は，ハードウェア記述と，検証の為の言語である．この言語は，新しい構文を新しく定義したわけではなく，Objective Caml 上のライブラリとして実装されている．Objective Caml で記述されたプログラムをコンパイルすると，様々な HDL や検査系用のコードを生成するための実行ファイルが生成される．HDCaml は Objective Caml として記述するため，Objective Caml のライブラリが，そのまま利用（インクルード）可能な点が特徴である．

† 信州大学大学院工学系研究科

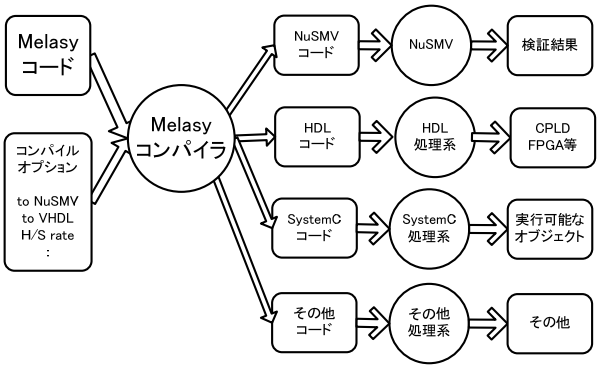

図 1 Melasy コンパイラ処理系の位置づけ

2.2 Confluence

Confluence[13]もハードウェア記述と，検証の為の言語である．この言語のコンパイラも Objective Caml で記述されているが，このコンパイラは，完全に独立したコンパイラとして動作する．Confluence では中間コードを用いたコード生成を行っており，Confluence コードは，前処理プログラムによって，まず中間コード(NetList) の状態へコンパイルされる．次に，後処理プログラムによって，NetList から VHDL や Verilog 用の対象コードを生成する仕組みとなっている．

3. Melasy コンパイラの位置づけ

Melasy は既存の様々な言語の上位言語として設計された．Melasy で記述されたハードウェア設計の記述は，Melasy コンパイラにより，様々な HDL による記述を生成する．Melasy コンパイラと SMV，HDL の関係について，図 1 に表す．Melasy コンパイラから，ハードウェア記述である Verilog-HDL や VHDL によるハードウェア記述を生成することで，通常のハードウェア設計を行うことが可能である．コンパイルオプションの変更で，NuSMV 等のモデルチェック用記述を生成することで，単一の Melasy コードから検証用のコードと，実行用のコードの両方を得ることができる．更に，ソフトウェア・ハードウェアの協調設計時においては，コンパイルオプションの切り替えにより，ハードウェア記述とソフトウェア記述の割合を指定することで，ハード・ソフトそれぞれの担当する処理の割合を変更した様々なコードを，単一の Melasy コードから生成が可能である．

4. Melasy 上位言語の設計方針と特徴

4.1 言語設計の方針

ハードウェアは，多くの場合同期をとりながら動作をするため，手続き型言語によるソフトウェアの設計で頻繁に行われる，繰り返しによる処理が非効率なものとなるケースが多い．ハードウェアでは並列に処理を行う設計上の工夫が頻繁に採用される．その結果，極めて酷似した記述（モジュール）が設計中に現れ，コードの可読性や記述性を大きく損なっている．本研究で提案した Melasy 言語では，繰り返しを表す構文により，一つの式から複数の式を生成し，このような問題の解決を目指す．また，式の再帰的な定義だけでなく，オブジェクトの定義についても再帰的な定義を行うことを可能にすることで，より複雑な設計の記述を，簡単に行うことが可能になるようにする．

Melasy をハードウェア記述だけでなく，将来的に SystemC 等による，ソフトウェア部分の設計も記述可能にすることによるメリットは既に上述の通りである．しかしながら，Melasy を通常の手続き型言語として設計すると，ハードウェアのコードを出力する際の並列処理部分の同期の扱いなどの実装が，複雑になってしまうため，この機能は採用しなかった．Melasy はオブジェクト指向の関数型言語を採用している．Melasy では，オブジェクトは，入出力をもつ回路となる．オブジェクトの入力の定義はインスタンスを宣言する時の引数として，出力はメンバ変数が担当する．それぞれのオブジェクトはハードウェアの回路に相当し，オブジェクトのインスタンスを作る際に入力と出力を結線する．また，既存の HDL による設計で多く見られるような，同じコードの単純な繰り返しは，コードの繰り返しでなく，繰り返しを表す構文によって記述する．このことで，必要なコード量の削減とともに繰り返しの構造の表す意味を明示化させる．

4.2 型推論

型推論とは，設計者（ユーザ）が明示的に対象オブジェクト（変数）の型を指定していない場合でも，コンパイラが型を推論することにより，正しい関数の定義を得るための仕組みである．コンパイラは型の判明している箇所より，接続や代入を介して型の推論を行う．型推論により，ユーザは場面における型を考えたり確認する手間を省略できるほか，曖昧な定義のコンポーネントを定義した上で，型推論の際に曖昧な定義から型の確定されたコンポーネントを作成することができる．また，さまざまな型に確定されるコンポーネントは，具体化された様々な物を出力することで，型に依存しないコンポーネントの定義が可能になる．これにより，あらゆる型に対して利用可能な演算器やセレクタのような物を定義することが可能になる．

4.3 記述性の比較

Melasy と SMV の記述性の違いを比較する．

4.3.1 曖昧なコンポーネント

Melasy ではテンプレートを使った曖昧なコンポーネントの定義により，ひとつの定義から様々なコンポーネントの定義をすることができる．例として，サイズが 2 のバッファとサイズが 4 のバッファを定義する場合は以下の通り．

[A] NuSMV による記述

```
Module buffer2()
VAR
  buf : array 0..1 of boolean;
Module buffer4()
VAR
  buf : array 0..3 of boolean;
Module main
VAR
  buf2 : buffer2;
  buf4 : buffer4;
```

NuSMV の場合，サイズ 2 のバッファとサイズ 4 のバッファを別々に定義する必要がある．

[B] Melasy による記述

```
component Hoge()
var
    hoe[N=8] :: Bool | N==0 | N==7  =1,
                     | otherwise    =0;
```

しかし，Melasy の場合は，曖昧なサイズのバッファをあらかじめ定義しておき，インスタンスを宣言する際に大きさを指定することで，ひとつの定義から様々なサイズのバッファを作ることができる．

4.3.2 配列の初期化

代入配列の初期化や代入の仕方を比較する．例として配列の先頭と末尾で特殊な初期化を行う場合を比較してみる．特殊な初期化の代わりに今回は先頭と末尾のみ 1 で，他を 0 で初期化を行うことにする．

[A] NuSMV の場合

```
MODULE Hoe
VAR
  hoe : array 0..7 of boolean;
ASSIGN
  init(hoe[0]):=1;
  init(hoe[1]):=0;
  init(hoe[2]):=0;
  init(hoe[3]):=0;
  init(hoe[4]):=0;
  init(hoe[5]):=0;
  init(hoe[6]):=0;
  init(hoe[7]):=1;
```

NuSMV では配列を扱うための構文がほとんど用意されていないので，全ての要素に対して初期化を行う必要がある．

[B] Melasy の場合

```
component Buffer<N>()
var
  buf[N] :: Bool;
component Main
var
  buf2 :: Buffer<2>;
  buf4 :: Buffer<4>;
```

Melasy の場合，ガード部と暗黙の foreach を使うことで簡潔に書くことができる．また，短く記述することでユーザの負担を減らすだけでなくコードを読む立場で考えた場合もより意味がはっきりとした記述となる．

5. 構文規則

5.1 構文規則

Melasy のコードは，基本的には回路を構成する "コンポーネントの定義の列"によって構成される．それぞれのコンポーネントの定義には，コンポーネントに含まれる変数を定義する var ブロックと，その状態の更新を定義する assign ブロックが含まれる．いずれのブロックでも式の記述にはガード部を持つ暗黙な foreach を使うが，これについては 5.2 で詳しく述べる．Melasy の文法を拡張 BNF で記述したものを，図 2 に示す.

5.2 ガード部付き暗黙な foreach

暗黙な foreach は, 配列の値を定義するのを支援するため仕組みである．これはひとつの定義式を，配列の全ての要素に対して作用させることで，ひとつの式で配列の要素の全てを定義することが可能となる．例えば，配列の何番目の要素を定義しているのかという情報は，特殊な定数を介して式に与えられるため，配列の部分情報によって，式の値を変化させることが可能である．また，ガード部を設けることで，式の形そのものを条件によって変えることも可能となる.

既存の HDL の多くは，文の繰り返し自体を表現する機能を有していないため，単純な繰り返しをシンプルに記述することができない．そこで，定数名や変数名といった識別子の文字列の一部を少しだけ変えながら記述を繰り返すような場面においても，いわゆるコピーアンドペーストや，簡単なプリプロセッサを自作することで同じような表現を作ることになり困難さ生じている．Melasy では暗黙な foreach とガードを用いることでこのような作業を減らすことができる．暗黙の foreach により配列の初期化の記述が 1 行で完了し，配列の一部に特別な値を入れるといった処理も，ガード部によって簡単に行うことができる.

SMV と Melasy による記述例の比較

[A] SMV の場合

```
next(reg[0]):= 1;
next(reg[1]):= 0;
next(reg[2]):= 0;
next(reg[3]):= 1;
```

[B] Melasy の場合

```
reg[N]| N==0||N==3 = 1,
      | otherwise  = 0;
```

このように，簡単に記述可能なほか，構造の意味，この場合配列の両端が 1 であり，それ以外は 0 である，といった情報を明示的に表すことができる

5.3 テンプレートつきコンポーネント

テンプレートはコンパイル時に処理される定数を，ユーザがインスタンスの宣言の際に自由に指定するための仕組みである．例えば，幅 4bit のバッファの定義と幅 8bit のバッファを定義を別々に定義するのではなく，テンプレートを用い幅 Nbit のバッファの定義を行い，インスタンスを生成する際に N に具体的な値を指定することで，一つの設計から様々な幅のバッファの設計を作ることが可能になる.

```
<idunit> = <lower> | <upper> | <digit>
<idunits> = <idunit>|<idunit> <idunits>
<id> = <lower> | <lower> <idunits>
<Id> = <upper> | <upper> <idunits>
<number> = <digit> | <digit> <number>
<varId> = <id> *<arraySize> ["." <varId>]
<suffixReferer> = "@" <number>
<substitution>= <substitution_> | <connection> |
1#<guard>
<substitution_> = "=" <expr>
<connection> = "(" #<expr> ")"
<if>="if"<expr_b> "then" <expr> "else" <expr>
<guard> = "|" <expr_b> <substitution>
<assign> = <varId> <substitution>
<program> = <components>
<components>=<component> | <component> <components>
<component> = "component" <Id> [<templateVar>] "("
[<arguments>] ")" *<block>
<templateVar> = "<" #<Id>">"
<templateArg> = "<" #<Expr>">"
<arguments> = #<argument>
<argument> = <varId> [<varType>]
<arraySize> = "[" [[<id> "="] <expr>]"]"
<block> = <varBlock> | <assignBlock> | <defineBlock>
<varBlock> = "var" *(<var> ";")
<var> = <varId> [<varType> [<templateArg>]
[<substitution>]
<varType> = "::" ("Bool" | <Id>)
<assignBlock> = "assign" *(<assign> ";")
```

図 2 Melasy 構文規則（拡張 BNF）

Melasy による記述例

```
component Buffer<N>()
var
  buf[N] :: Bool;
component Main
var
  buf2 :: Buffer<2>;
  buf4 :: Buffer<4>;
```

6. モデル検査ツール向けコード生成

Melasy の暗黙な foreach やテンプレートを含むコードと，モデル検査ツール SMV で記述可能なコードの間には，表現力に大きな違いがある．そのため，Melasy のコードから SMV のコードを生成する際には，まず最初に暗黙な foreach とテンプレートを展開し，これらの機能が含まれない中間コードを生成することとした.

6.1 中間コードの生成手順

[step1] 暗黙な foreach の展開

暗黙な foreach の展開では，配列の添え字を特殊な定数に置き換え，定数リストにその定数と添え字を表す数値に置き換えている.

[step2] テンプレート の展開

テンプレートの展開では，テンプレート引数に渡された値を使ってテンプレート名を展開（mangle）し，展開されたコンポーネント中に含まれるテンプレート引数は，全て定数に置き換えられる．今回開発したコンパイラでは，名前マングルは名前の後ろに，テンプレート引数に入れられた値を，アンダーバーで区切って追加している．また，テンプレートを含むコンポーネント内で定義されているインスタンスの宣言には，テンプレート引数が使われている場合があるので，テンプレートの展開が行われなくなるまで，同じ処理を繰り返す必要がある.

この段階で曖昧なコンポーネントは具体化されており，暗黙な foreach も一般的な配列への代入へ置き換えられている．ただし，一部の構文については簡単な変更を加える必要がある.

[step3] if then else 構文の展開

SMV には if then else 構文は存在しない．代わりに case～esac による値場合分けの機能を有するため，条件を複数の case 文に置き換えている．

[A] Melasy の条件文

```
if expr then a else b;
```

[B] SMV の case 文への展開

```
case
  expr : a;
  1    : b;
esac;
```

[step4] 配列 の展開

今回開発したバージョンの Melasy コンパイラでは，多次元配列は扱えない．将来的は多次元配列を扱う予定で開発を進めている．SMV では多次元配列には対応していない為，これは一次元配列に展開する必要がある．例えば、以下のような記述を行うことができない。

```
next(value[0]) = value[n];
```

SMV では，配列の添え字に変数を使ったアクセスが許されていないので value[0]を value_0 のような識別子で扱ったとしても問題が生じない．従って、全ての配列を添え字付与した識別子の群として扱うことで，多次元配列への対応を行う．

6.2 SMV コードの生成例

以下に Melasy コードから SMV のコードを生成した例を取り上げる．この例では，Sample コンポーネントとして，Boolean 型の配列 values[] を長さ LEN だけ生成し，そのインスタンスは初期状態が values の index の順で設定され，更に values の次段ステップの更新値として，(index 値 - 1) mod (LEN-1) の値をストアするような，単純な規則によって構成されるステートマシンを定義している．そして，Main コンポーネントにおいて，この Sample コンポーネントの LEN= 4 によるインスタンス生成を行っている．具体的には幅 4 のローテーション型レジスタを実装したものとなっている．

[A] Melasy コード

```
component Sample<LEN>()
var
   values[LEN] :: Bool = @1;
assign
   values[] | @1==0     = values[LEN-1],
            | otherwise = values[@1-1] ;

component Main()
var
   likeThis::Sample<4>();
```

[B] 生成された SMV コード

```
MODULE Sample_4()
VAR
   values_0 : boolean;
   values_1 : boolean;
   values_2 : boolean;
   values_3 : boolean;
ASSIGN
   init(values_0) := 0;
   init(values_1) := 1;
   init(values_2) := 2;
   init(values_3) := 3;
   next(values_0) := values_3;
   next(values_1) := values_0;
   next(values_2) := values_1;
   next(values_3) := values_2;
MODULE main()
VAR
   test : Sample_4;
```

7. 評価と今後の課題

テンプレートを使うことで柔軟に利用できるコンポーネントを定義することが可能になった．また，簡単にインスタンス時のサイズ変更が可能になるため，モデル検査ツールの様な，非常に時間的コストのかかる処理を行う場合等に，まず最初に問題のサイズを一時的に小さくした SMV コードを出力してモデルチェックを行うことで，設計の初期段階におけるモデル検査に要する時間と書き直しの手間など，様々なリソースを抑えることができるようになった．小さい問題のサイズにおいてモデル検査を通過した後，実サイズの問題として SMV コードを最終的に出力してモデル検査を実施する．

暗黙な foreach とガード部によって，複雑な初期化を代入文の列挙ではなく，場合分けによる記述に置き換えることが可能となった．これにより，コードを記述する量が減るだけでなくコードの表す意味をより明確に表すことができるようになった．

今後は，ユーザ定義のライブラリと型の設計を容易にする工夫を行うことで，Melasy の記述力を高める為のライブラリの開発を簡単に行うことが可能になると考えられる．

Melasy コンパイラは，現在 SMV のコードが出力できるが，今後はより多くの HDL のコードを生成可能にする目標が．様々な言語のコードを生成可能にするにあたって，Melasy コードから直接生成するコードを，Melasy 中間コードの一つに限定し，このコード生成の段階で，暗黙の foreach の展開や，テンプレートの展開を行うことで，既存の言語と近い機能しか持たない上位言語を作ることができると考える．

参考文献

[1] V. エイホ , R. セシィ , J. D. ウルマン：コンパイラ　原理・技法・ツール 1,2：サイエンス社(1990)

[2] 中田 育男：コンパイラ：産業図書(1981)

[3] 中田 育男：コンパイラの構成と最適化：朝倉書店(1999)

[4] 佐々 政孝：プログラミング言語処理系：岩波書店(1989)

[5] VHDL(VHSIC Hardware Description Language)：http://vhdl.org/

[6] SystemC：http://www.systemc.org/

[7] E. M. Clarke, Orna Grumberg, Doron Peled：Model Checking：Mit Press(2000)

[8] SMV(Symbolic Model Verifier)：http://www.cs.cmu.edu/~modelcheck/smv.htm

[9] NuSMV：http://nusmv.irst.itc.it/

[10] Haskell：http://www.haskell.org/

[11] Parsec：http://www.cs.uu.nl/people/daan/download/parsec/parsec.html

[12] HDCaml：http://www.confluent.org/wiki/doku.php/hdcaml

[13] Confluence：http://www.confluent.org/wiki/doku.php/confluence