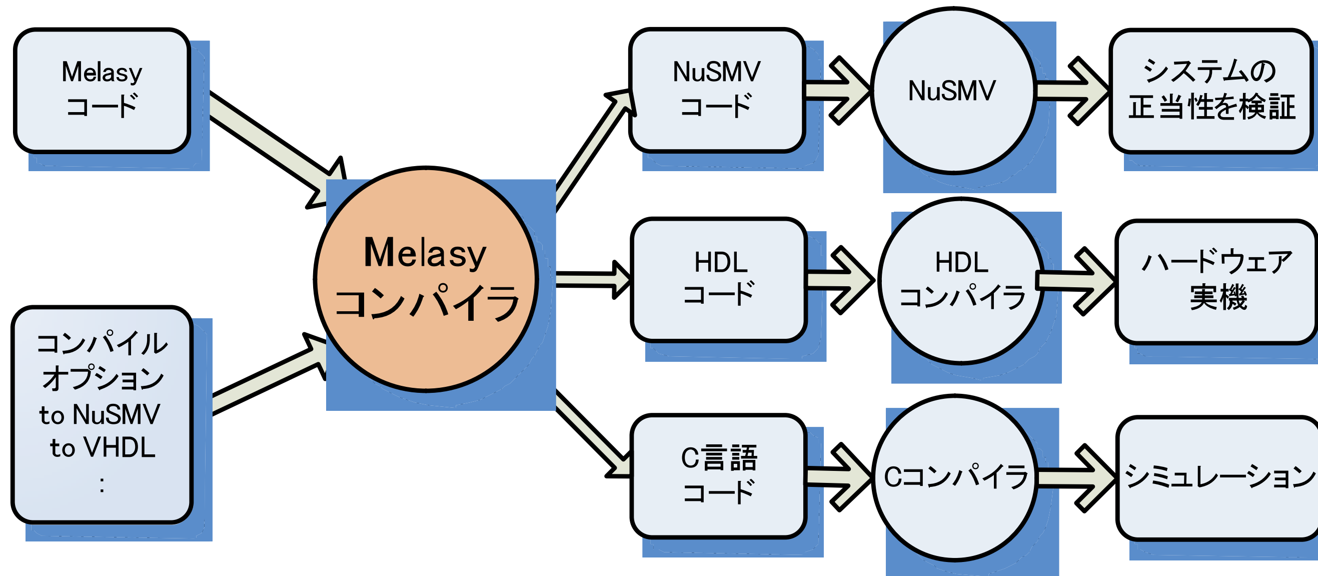
モデル検査に対応する上位ハードウェア記述言語 ”Melasy”の設計と実装

信州大学大学院 工学系研究科 情報工学専攻
岩崎直木

Melasyの概要

- 既存のハードウェア記述言語の上の層に新たなハードウェア記述言語を置く.



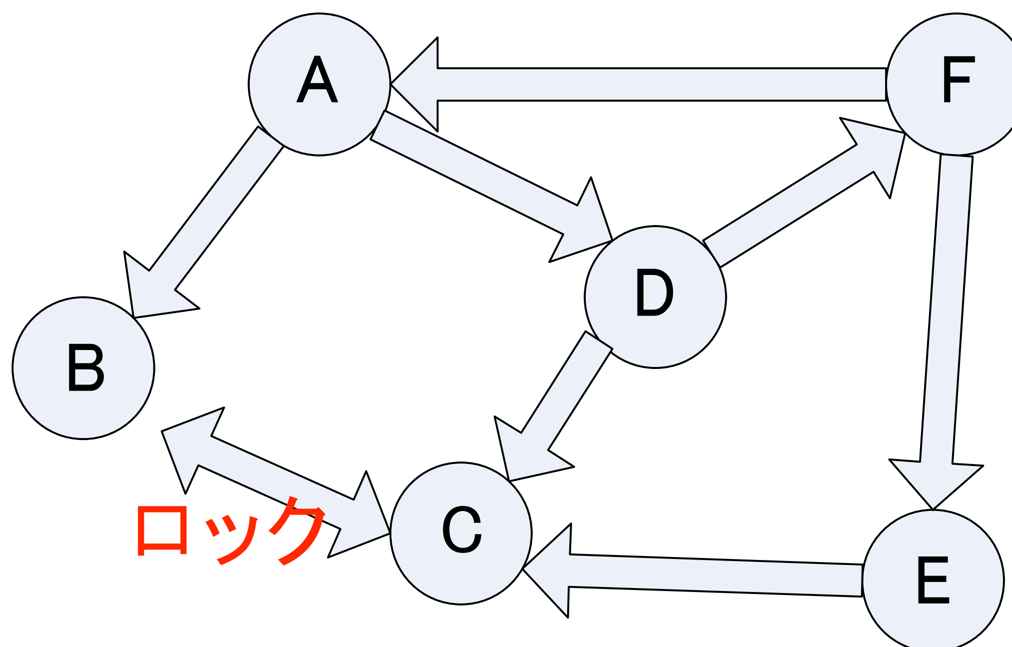
- 1つのMelasyコードが様々な環境で動作する.

背景

- バグのない設計は困難
- 設計のバグを見つけるのも困難
  デッドロックしないか．．．
  仕様から外れた動きをしないか．．．
- しかしシステムの信頼性は要求される

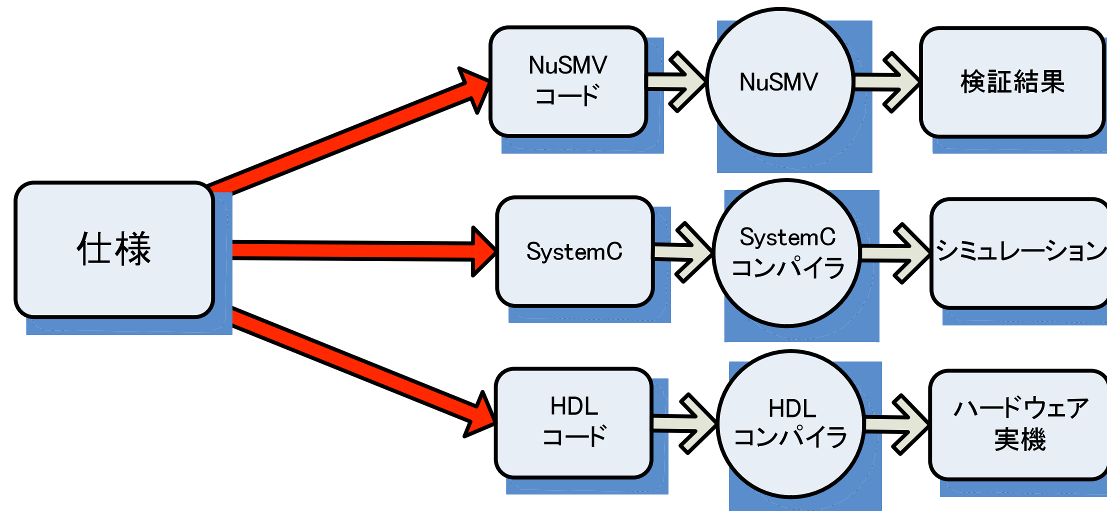
モデル検査

- システムの動作を専用の言語で記述
- システムの満すべき条件を定義
- システムのなりうる状態を網羅的に検査



コードを記述

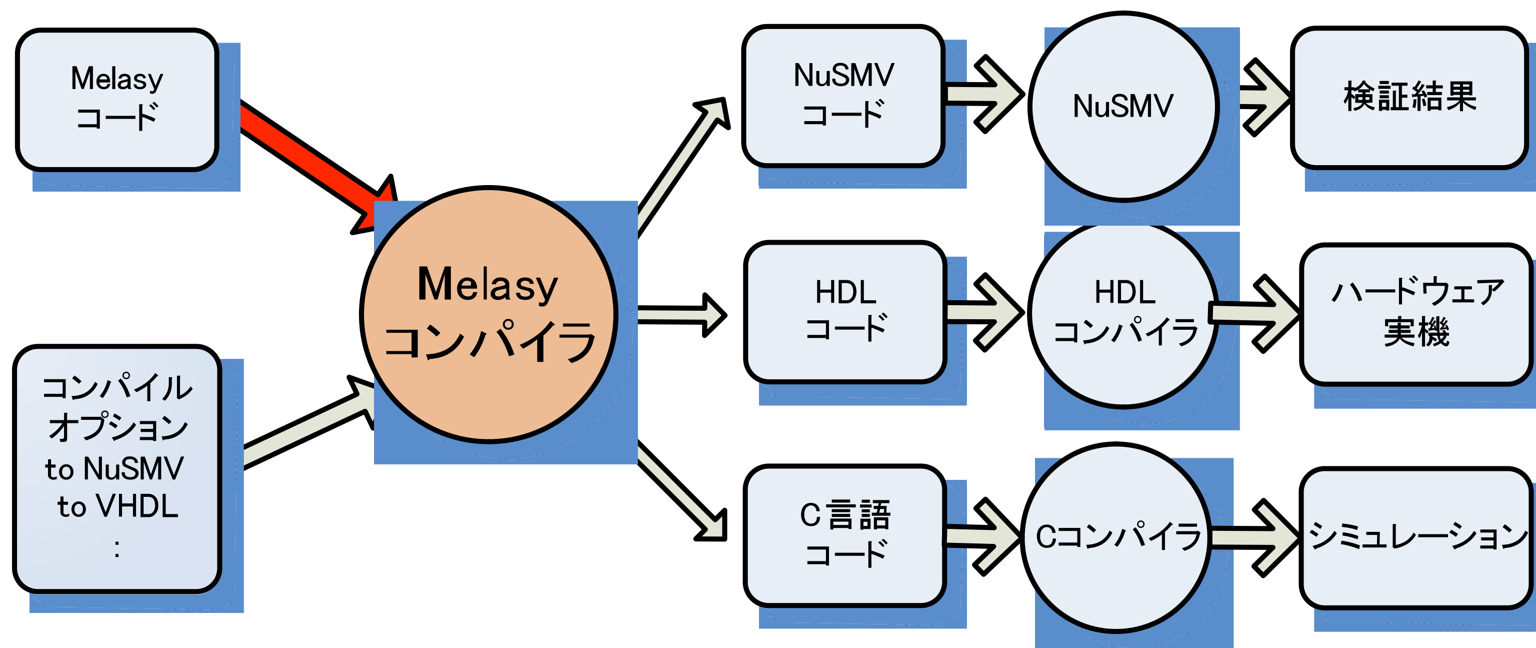
- 仕様にバグがないか？ → モデル検査を行う
- 正しいらしい　→ 回路設計，シミュレーション
- 駆動した　→ 実機向けのコードの記述



- 仕様に変更が起きたら → ?!?!

目的:仕様からコードを生成する作業を自動化

- 既存の環境向けコードを吐くコンパイラの開発

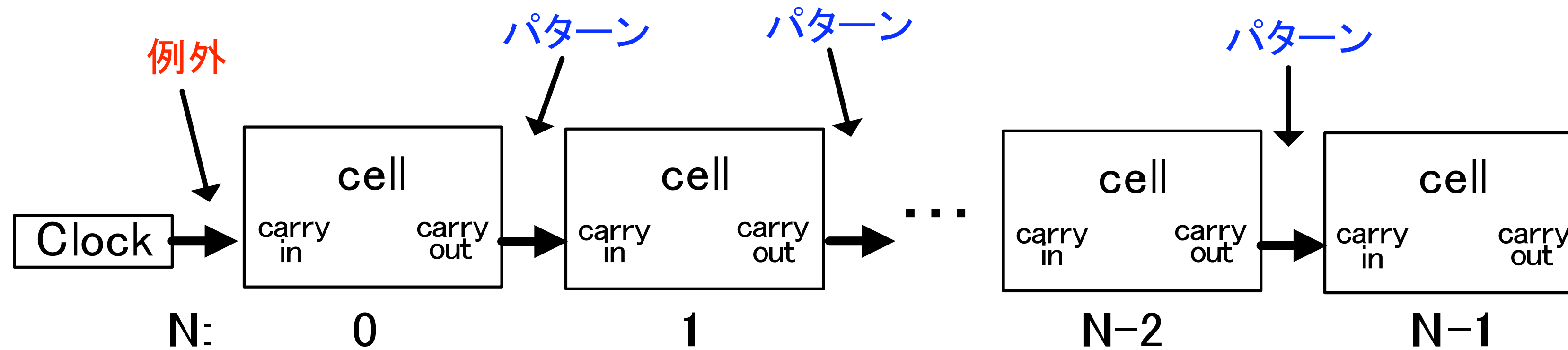


- 各環境毎にコードを記述する必要がなくなる

初期のMelasyの言語仕様

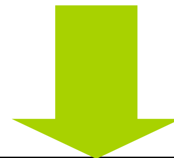
- 様々な環境向けにコードが吐けても
  非力な言語では意味が無い
- ガード式(ガード条件)付き暗黙のforeach
  - ハードウェアは並列動作が基本
  - 似たような記述の連続
  - これを効率的に記述
- テンプレート
  - 再利用のしやすいコードを記述

ガード式付き暗黙のforeach




```
・・・略

cells[N] :: CounterCell | N==0       (clk),
                        | otherwise  (values[N-1].carryOut);
```



```
MODULE Counter_4(clk)
VAR
        cells_0 : CounterCell(clk);
        cells_1 : CounterCell(cells_0.carryOut);
        cells_2 : CounterCell(cells_1.carryOut);
        cells_3 : CounterCell(cells_2.carryOut);
```

テンプレート

- バッファのサイズ等を曖昧にした記述が可能
- 様々な状況で再利用が可能になる

```
component Buffer<N>()
var
    buf[N] :: Bool;

component Main()
var
    buf4 :: Buffer<4>;
    buf8 :: Buffer<8>;
```

旧Melasyの欠点

- ガード式とforeachでは記述できない例

```
out[0] = value[0]
out[1] = value[0] & value[1]
out[2] = value[0] & value[1] & value[2]
out[3] = value[0] & value[1] & value[2] & value[3]
```

```
out[N] = value[0] & ... & value[N] ?????
```

- 分割コンパイルの非対応
- 抽象化の機能がコンポーネント単位のみ
- コード生成の難しさ
  テンプレートなどを含む複雑な構文木から
  各言語向けにコードを生成する必要があった.

新Melasy

- 言語仕様，コンパイラの構造を見直す．

旧Melasyの欠点

- ガード式とforeachでは記述できない例

```
out[0] = value[0]
out[1] = value[0] & value[1]
out[2] = value[0] & value[1] & value[2]
out[3] = value[0] & value[1] & value[2] & value[3]
```

```
out[N] = value[0] & ... & value[N] ?????
```

- 分割コンパイルの非対応
- 抽象化の機能がコンポーネント単位のみ
- コード生成の難しさ
  テンプレートなどを含む複雑な構文木から各言語向けにコードを生成する必要があった.

コード生成器開発の難しさ

- 対象言語向けコード生成器に渡す構文木はテンプレート等を含むもの
- NuSMV用コード生成器，VHDL用コード生成器．．．それぞれがテンプレートの展開を行ったりする必要があった

中間言語を介したコード生成

- 中間コード生成器がテンプレートなどを処理し，中間言語はシンプルな構造にする.
  - ➔コンポーネント
  - ➔入出力
  - ➔動作の定義
- 各言語向けコード生成器は，難しい処理を行う必要が減る
- XMLで記述 → ライブラリが揃っている

NuSMV向けコード生成器の開発

- 新Melasy
  - ➔コード生成部 144行(Python)
    - component -> module
    - 式の構文木 -> 式
    - ．．．．
- 旧Melasy
  - ➔コード生成部 590行(Haskell)
    - テンプレートを展開
    - コンポーネントの依存関係を解決
    - ．．．．．．．．

言語仕様を見直す

- ガード式とforeachでは記述できない例

```
out[0] = value[0]
out[1] = value[0] & value[1]
out[2] = value[0] & value[1] & value[2]
out[3] = value[0] & value[1] & value[2] & value[3]
```

```
out[N] = value[0] & ... & value[N] ?????
```

- 分割コンパイルの非対応
- 抽象化の機能がコンポーネント単位のみ
- コード生成の難しさ
  テンプレートなどを含む複雑な構文木から各言語向けにコードを生成する必要があった.

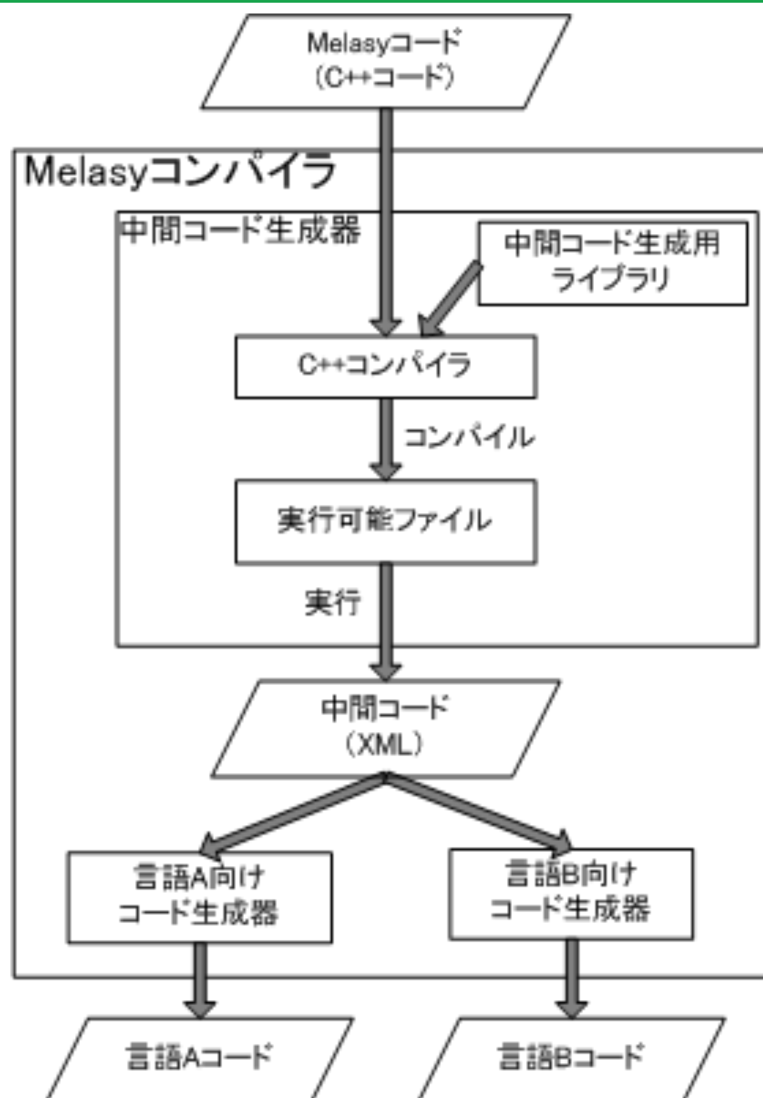
C++をプリプロセッサとして利用する

- C++でこれを標準出力に吐き出すだけなら楽

```
out[0] = value[0]
out[1] = value[0] & value[1]
out[2] = value[0] & value[1] & value[2]
out[3] = value[0] & value[1] & value[2] & value[3]
```

- C++を中間コード生成器として使えないか？

構造



C++上にハードウェア記述言語を乗せる

- ハードウェアの記述を行う為のライブラリを開発

1. ライブラリを利用してハードウェアを記述
2. コードの構文木を取得する

```
a = b & c;
```

→

```
(= a (& b c))
```

```
a = b[0];
for(int i=1; i<3; i++)
    a = a & b[i];
// a = b[0] & b[1] & b[2]
```

→

```
(= a (& b[0] b[1] b[2]))
```

複雑な式が，C++と同程度の効率で記述可能に

分割コンパイル・抽象化機構

- 分割コンパイルの非対応
- 抽象化の機能がコンポーネント単位のみ

- 他の旧Melasyコンパイラの欠点についてもC++の言語機能によって解決
  - ➔C++コードは分割コンパイルに対応
  - ➔関数により，より粒度の細かいコードが再利用できる

評価

- 旧Melasyで上位言語を開発
  - ➔一つのコードから，様々な環境向けにコードを生成
  - ➔言語としての記述力には問題あり
  - ➔対応言語を増やすためには，沢山の作業が必要
- 新Melasy
  - ➔旧Melasy以上の記述力を獲得
  - ➔対応言語の追加が容易に

問題点

- C++コンパイラは，記述にミスがあっても
  C++用のエラーメッセージを吐かない
- C++的にはOKでもMelasy的にはNGなコード
  コンパイル時にエラーを出すのが困難
- 独自の構文の追加が不可能
  新しい演算子や構文の追加ができない

まとめ

- 様々な環境で利用可能なMelasyコンパイラの開発
- 一つのMelayコードから，様々な環境で動作するコードを生成
- C++をプリプロセッサのように利用可能
- 新しい言語に対応させることは，比較的容易