設計自動化 85－23
（1997. 10. 29）

合成ディレクティブを組み込んだ動作レベル設計記述言語

舟本 一久　五島 正裕　森 眞一郎　富田 眞治

京都大学大学院工学研究科情報工学教室
〒606-01 京都市左京区吉田本町
TEL: 075-753-5393
E-mail: {funamoto, goshima, moris, tomita}@kuis.kyoto-u.ac.jp

あらまし

既存のHDLを用いたハードウェア開発では，設計の初期段階から物理的制約を勘案して設計記述することが要求されるため，トップダウン開発が困難となっていた．このような問題を解決するために，動作レベル設計記述言語「**Evelyn**」とその処理系を開発している．Evelynでは，モジュール間の境界や通信を抽象的に記述でき，設計者は動作設計部分に傾注することができる．また，パイプラインを抽象化し，複数ステージに渡る一連の動作を自然に記述することができる．さらに設計が物理的制約を満たさなかった場合はディレクティブを変更するだけでよい．

キーワード 動作レベルHDL，Evelyn，モジュール境界の抽象化，合成ディレクティブ

A Behavior Description Language with Built-in Synthesizer Directives

Kazuhisa Funamoto　Masahiro Goshima
Sin-ichiro Mori　Shinji Tomita

Department of Information Science,
Faculty of Engineering, Kyoto University
Yoshidahon-machi, Sakyo-ku, Kyoto 606-01, JAPAN
E-mail: {funamoto, goshima, moris, tomita}@kuis.kyoto-u.ac.jp

Abstract

Conventional hardware description languages (HDLs) are not suited for the top down design. It is because, from the very early design stage, they demand designers to consider the physical constraints and describe the designs carefully. Then we propose a new HDL **Evelyn**. With Evelyn, designers can describe the module boundary abstractly, so can concentrate on the architecture level design. And they can also abstract pipeline operations and describe multi-cycle behaviors naturally. Furthermore, if the design doesn't satisfy the physical constraints, designers only need to change the synthesizer directives.

key words　behavior description language, Evelyn, abstracted module boundary, synthesizer directive

1 はじめに

VHDL や Verilog-HDL といった機能レベルのハードウェア記述言語 (HDL) と，それらからの (半) 自動的な論理合成/論理最適化によるハードウェア設計手法が一般に普及しつつある．機能レベル HDL による設計手法によって，従来の回路図入力に対して，設計の効率は確かに飛躍的に向上した．

しかし，機能レベル HDL によっても当初言われた程に設計の抽象度が上がるということはなく，本当の意味でのトップダウン設計を可能とするには全く不十分であった．言語入力には回路図入力に比べて入力や修正が容易になるという利点があるが，機能レベル HDL の導入によって得られた効率化はむしろこのことによる部分が大きい．

設計の抽象度が上がらなかったのは，HDL の処理系 – 主に論理最適化系の能力が貧弱であったこともさることながら，機能レベル HDL の記述能力自体の低さにも大きな原因があった．従って，より抽象度の高い HDL が求められるのは自然なことであり，動作レベル (behavior level) と呼ばれる，より抽象度の高い記述をサポートする HDL とその処理系が幾つか提案されている．

動作レベル HDL は，従来の HDL より抽象度の高い記述を可能とするものであるが，もちろんただ単に抽象度を高めただけでは，そこからの合成 – 機能合成が困難となる．従って，動作レベル HDL の記述能力は，機能合成技術に制限されることになる．例えば，既に提案されている動作レベル HDL は，この制限をクリアするために記述の対象を限定する場合が多い．

現在我々の研究室で開発中の「**Evelyn**」も，既に提案されている動作レベル HDL やその処理系と同様に，抽象度の高い記述とそれからの半自動的な機能合成を目標とする．しかし Evelyn では，記述の対象を限定するのではなく，合成ディレクティブ (synthesizer directive) を言語の仕様として組み込むことで，抽象度の高さと合成能力のトレードオフを高いレベルでバランスさせるアプローチを採る．本稿では主に，その動作記述の方法について述べる．

以下では，まず 2章で Evelyn とその処理系を設計するにあたっての方針を説明する．そして，3章で Evelyn の動作記述について説明し，4章で Evelyn 記述の具体例を示す．最後に 5章で構想中の Evelyn の機能合成方法について触れる．

2 Evelyn の設計方針

本章では，まず既存の機能レベル HDL の設計の抽象化に関する問題点について考察し，次にその問題を解決するための糸口を与える．

2.1 機能レベル HDL の問題点

機能レベル HDL でも，抽象度の高い記述をサポートする必要性について考慮されていない訳ではない．例えば VHDL では，以下のような開発方法を想定し，サポートしている [1]:VHDL の各モジュールは複数の *architecture body* を持つことができる．設計者は，まず抽象度の高い *architecture body* を記述して設計検証を行ない，それを論理合成可能な抽象度の低いものに順次置き換えていく．しかし，このような能力が実際に設計の抽象化に対して有効に働く場面は限られている．以下ではその理由を述べる．

まず，論理最適化の能力には自ずから限界があることを述べておく．論理最適化は，本質的にオーダが大きい処理であり，入力する回路規模が大きくなるとその処理量は爆発的に増大する．最適化技術の発展とともに系の許容する回路規模は大きくなってはいくだろうが，実質的に限界が無くなるという訳ではない．従って設計者は，論理最適化系が設計制約を満たすことのできる回路の規模を常に考慮しておく必要がある．

さて，論理最適化能力には限界があるとした上では，既存の機能レベル HDL の，**モジュール境界の不撓性**が特に問題となる．それらの HDL では．モジュールの境界を変更するような設計の変更には多大な労力を要する．そのため，以下の設計コストの増大を避け難い．

問題-1. 初期工程のコスト

論理最適化系が設計制約を満たすことのできる回路の規模を勘案し，設計上クリティカルな部分回路を 1 つのモジュールにまとめあげることを重視して，設計の最初の段階から綿密にモジュールの分割を決定する必要がある．

その結果，ハードウェアの動作を自然に記述することは二の次にならざるを得ず，しばしば

- 意味的に関連性の高い記述を複数のモジュールに分割したり，逆に，
- 意味的に関連性の低い記述を 1 つのモジュールにまとめたり

する必要が生じる．この作業は人手で行わざるを得ず，アルゴリズム・レベルでバグが混入する可能性が高い．

問題-2. 設計変更のコスト

初期工程で綿密にモジュール分割を行ったとしても，論理最適化の段階で制約を満たさないことが判明することは完全には避け難い．その場合には，モジュー

ル間で記述の移動をやはり人手で行う必要がある．このような作業は，既に作り込まれた記述に対するものであるだけに，やはりバグが混入し易い．

本節の冒頭で述べたVHDLの抽象化の能力は，モジュール境界の不撓性をむしろ高めており，上述の問題点を全く解決しない．

2.2 Evelynの設計方針

2.2.1 モジュール境界の抽象化

前節で述べたモジュール境界の不撓性の問題を解決するためには，まず記述におけるモジュール境界と実際に論理合成/最適化系に入力されるモジュールの境界とを分離する必要がある．そのためには，記述において，モジュール境界が具体的過ぎてはならない．すなわち，**モジュール境界の抽象化**が必要である．

さて，動作レベルHDLでは，複数ステージに渡る一連の動作を自然に記述し，その記述から各ステージの機能レベル記述を（半）自動的に合成する．それに対し機能レベルHDLでは，対象が十分には小さくない場合，各ステージごとにモジュールを人手で分割しておく必要があるだろう．従って，動作レベルHDLにおける複数ステージに渡る一連の動作の記述は，複数ステージ間の境界の抽象化であるとすることができる．複数ステージに渡る一連の動作を抽象的に記述することによって，下流工程においてステージを機能合成する余地が生じるのである．

しかし一方で，既存の動作レベルHDLは，複数ステージに関する境界のみを抽象化しているに過ぎないことが分かる．そこでEvelynでは，ステージ間の境界に限らず，**一般のモジュール境界を抽象的に記述し，そこから論理合成/最適化に適した機能モジュールを人手を介さずに合成する**ことを最終目標とする．これによって，前節で述べた問題点は完全に解決される．

2.2.2 合成ディレクティブ

しかし，複数ステージの場合とは異なり，一般のモジュール境界の抽象的な記述から，論理合成/最適化に適した機能モジュールを自動的に合成することは格段に難しい．完全に自動的に合成することを目指すアプローチは，前述の論理合成の限界の議論と明らかに矛盾する．

従ってEvelynでは，抽象的なモジュール境界の記述から論理合成/最適化に適した機能モジュールを合成するに際して，プログラマからの**合成ディレクティブ**を要求するアプローチを採る．

すなわちEvelynは**動作記述**と**合成ディレクティブ**の2つの要素からなる．合成ディレクティブは言語仕様の一部として規定する．動作記述のみでシミュレーションによる設計検証が可能であるが，それだけで合成可能である必要はない．動作記述に対して，十分な合成ディレクティブを与えることで，はじめて機能合成が可能になればよい．なお，合成ディレクティブに関する変更は，記述されたハードウェアの動作に性能以外の点で変化を及ぼさない．従って，このアプローチは，人手を介さない合成というEvelynの最終目標と矛盾することはない．

本節で述べた最終目標が達成されれば，Evelynは，動作設計と言わず，方式設計の下流工程あたりから使っていける可能性がある．

3 動作レベル設計記述言語 Evelyn

本章では，以上述べた方針を基に開発している動作レベルHDL Evelynの動作記述について概要を述べる．なお，具体例は次章で示す．

3.1 Evelynが提供する抽象化能力

Evelynは，以下の抽象化をサポートする:

データの抽象化 列挙型，多次元の配列型，構造体など，抽象度の高いデータ型をサポートする．それらは(半)自動的にbit列に合成される．

モジュール境界の抽象化

パイプラインの抽象化 複数のステージに渡る一連の動作を自然に記述し，その記述からパイプラインを(半)自動的に合成する．

その他のモジュールの抽象化

これらの抽象化のうち，本稿では特にモジュール境界とパイプラインの抽象化に焦点を当てて説明する．

3.2 動作記述の概要

Evelynでは，モジュールを記述の単位とする．ここでいうモジュールは機能モジュールとは関係ない．あくまで意味的な塊である．機能合成時に合成ディレクティブによって，モジュールを自在に分割/統合して物理的回路へマッピングすることができる．動作を記述しやすいようにモジュール構成を設計すればよい．

モジュールが行なう一連の処理をそれぞれ**メソッド**として記述する．モジュール間の通信は，相手のメソッドを起動することで実現する．つまり，Evelynでは通信はメソッド呼び出しに抽象化される．

メソッドには複数サイクルに渡る動作も記述できる．時間方向に記述できるので，動作記述は非常に素直なものとなる．確かに時間方向を主体に記述すれば時間に垂

直方向の処理が記述しづらくなるが，後に述べる抽象化のため，そのような記述も困難なく行なえる．

Evelyn では，複数のモジュールが互いにメソッドを起動し合いながら動作することで，ハードウェアをモデリングする．

3.3 実行モデル

メソッド呼び出しによって行なわれる1回のメソッドの実行を**プロセス**と呼ぶ．

複数サイクルに及ぶプロセスは，いくつかの**ステージ**に分割される．Evelyn では，このステージが実行の単位となる．

実行中の各プロセスの1つのステージは，**サイクル**毎に同時に実行される．また，ステージ中の命令は逐次的に実行されるが，それらはδ時間で実行されるものとする．つまり，実行途中のステージの状態を外から観測されることはない．

詳しくは3.5節で述べるが，ステージに分割されたプロセスは基本的にパイプライン実行可能である．Evelyn では特に指定しない限りこれらをパイプライン実行する．

3.4 メソッド

3.4.1 メソッド呼び出し

Evelyn のメソッドが一般の関数などと異なる点は，Evelyn のメソッドには，多サイクルを要する処理も記述できるという点である．

メソッド呼び出しの後，呼び出したプロセスはメソッドの完了を待たずに後続命令の実行を続け，返り値が必要になった時点で実行をブロックする．

呼ばれたプロセスは，返り値が用意できた時点で呼び出しプロセスに値を返す．返り値が複数ある場合には，それらが別々に返されることがある．

3.4.2 メソッドの同時呼び出し

一般のプログラミング言語と異なり，ハードウェア記述言語の場合には，同一モジュールに対してメソッドが同時に呼び出されたときにはそれらを統一的に処理する必要がある．

Evelyn ではそのような場合の処理を restricted method と呼ぶ特別なメソッドを記述する．restricted method は，同一サイクルで自分を呼び出した全てのプロセスに対して，総合的に処理を行なう．

詳しくは，4.2節にて例を挙げて説明する．

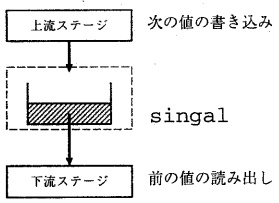

図 1: singal

3.5 暗示的なステージ分割

AIDL[2], BDL[3] など明示的にパイプライン・ステージを定義するアプローチと異なり，Evelyn では，特殊な変数への書き込み/ 読み出しという半順序関係によって，パイプライン・ステージを暗示的に定義する．

半順序関係から暗示的にパイプラインを規定するため，「この処理が始まるまでに…」，「ここからここまでの間に…」などという記述ができ，タイミングに関する記述の自由度が高い．

本節では，まずその特殊な変数について説明したのちに，Evelyn におけるステージ分割について説明する．

3.5.1 signal と singal

プロセスは，次の2種類の変数によってステージに分割される．

- **signal** : 信号線に相当．
- **singal** : フリップフロップ (FF) に相当．

まず，これらの変数について説明する．

signal signal とは，実際のハードウェアでいうところの信号線に相当する．signal はサイクル間で値を保持しない．

singal サイクル間で記憶を保持しない signal に対して，保持するのが singal である．しかし Evelyn では，singal は単に FF を表現するだけでなく，プロセスをステージに分割する役割も持つ．

singal とは直感的には長さ 1 の FIFO のようなもので，以下のような性質を持つ変数である (図 1参照)．

- 書き込み操作と読み出し操作を同時に行なえる
- 前の値を参照している process が存在する間は次の値を書き込むことはできない

このような特徴を持つ singal を用いると，パイプラインを簡単に記述することができる．すなわち，

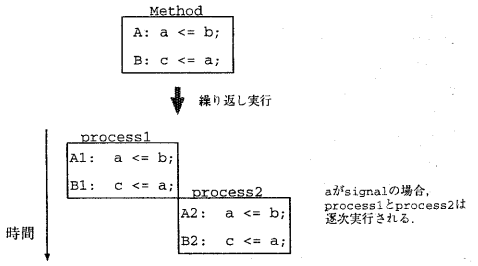

図 2: signal 変数

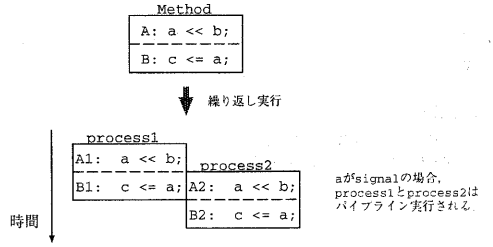

図 3: singal 変数

- 下流ステージで値を参照しながら，上流ステージは値を生産することができる．
- 下流ステージが何らかの原因でブロックした場合には，上流ステージも次の値を生産することができず，実行をブロックされる．
- 上流ステージが何らかの原因で次の値を生産できなかった場合には，下流ステージの実行はブロックされる．

ハードウェアの動作を時間方向に普通に記述すると，有効ビット，acknowledgement 信号などの処理 – 特にディアサートが非常に煩わしい．何もしていないときの動作を記述しなければならないからである．

Evelyn では，singal が本質的に持つ同期機能のため，このような基本的な同期を暗示的に記述することができる．そのため設計者は基本的な同期の記述に頭を悩まされず，より高レベルの設計に専心できる．

3.5.2 ステージ分割

ステージは，singal によって以下のように定義される．

定義 singal の参照から，singal への書き込みまでの一連の処理をステージという．

それでは，singal によってステージ分割される様を，例を挙げて説明する．

例 1：signal のみで記述されたメソッド まず，図 2のようなメソッドを繰り返し実行する場合を考える．図中，<=は signal への代入を表す．

この例では，プロセス全体が 1 つのステージとなる．このため，結果的に process1 の終了を待って process2 が実行されることになる．

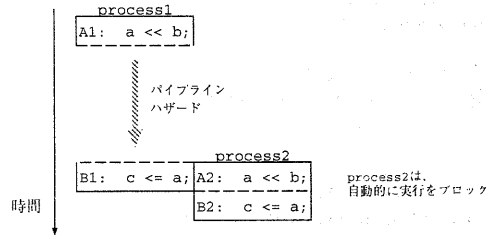

図 4: パイプライン・ハザード

例 2：singal によるステージ分割 次に，図 2のメソッドの a を singal に変更してみる (図 3)．図中<< は singal に対する代入を意味する．a が singal になったことで，A と B の間でステージ分割されることになる．

singal は書き込みと読み出しが同時に行なえるので，B1 と A2 はオーバラップして実行されるようになる．これは，process1 と process2 がパイプライン実行できるようになったことを意味する．

例 3：パイプライン・ハザード 図 3のメソッドを実行中に，図 4のように，process1 に何らかのハザードが生じた場合を考えてみる．この場合，Evelyn では，自動的に process2 の実行がブロックされる．

これは，singal の「前の値を参照しているプロセスが存在している限り，次の値を書き込むことはできない」という性質のためである．

このように，ある変数を signal から singal に変更するだけで，そこにパイプライン・ラッチを挿入することができる．

4 Evelyn の動作記述例

本章では，Evelyn の動作記述例を示す．

```
module processor{ /* モジュールの定義 */
  private:
    :
  public:
    void pipeline( void );
    boolean bus_req( void );
      :
}

void processor::pipeline( void ){ /* メソッドの定義 */
  /* if stage */
  ir.write ( mem.read( pc.read() ) );
  pc.inc();

  /* decode stage */
  operation.decode( ir );

  /* execution stage */
  if ( operation.IsEX() = true ){
      :
  }

  /* branch stage */
  else if ( operation.IsBR() = true ){
      :
      pc.write( newpc );
  }
    :
}
```

図 5: パイプライン・プロセッサ記述例

```
module pc{ /* モジュールの定義 */
  private:
    singal int _pc;
    restricted_method core( int newpc );
  public:
    int read( void );
    void inc( void );
    void write( int newpc );
}

void pc::write( int newpc ){ /* 書き込みメソッド */
  core( newpc );
}
void pc::inc(){ /* インクリメント */
  core();
}

/* restricted method の定義 */
restricted_method pc::core( int newpc ){

  if ( write in CALLER ){
    _pc << newpc;
    for write return SUCCESS;
    for inc return CANCEL;
  }
  else if ( inc in CALLER ){
    _pc << pc+1;
    for inc return SUCCESS;
  }
}
```

図 6: restricted method の例

4.1 パイプライン・プロセッサ

Evelyn では，プロセッサ，メモリなどのオブジェクトをモジュールとして記述し，モジュールに対する一連の操作をメソッドとして記述する．

図 5は，簡単なプロセッサの記述の一部である．プロセッサをモジュールとして記述し，命令パイプラインをプロセッサの 1 つのメソッドとして記述する．

図から分かるとおり，この段階ではただ処理の流れを素直に記述するだけである．実際の論理回路へのマッピングを考慮する必要はない．

図 5中，pc.read()，pc.inc() などの記述はモジュール pc(プログラムカウンタ) のメソッドの呼び出しである．モジュール間の通信は，このようにメソッド呼び出しによって実現される．

4.2 プログラムカウンタ

例えば，図 5のプロセッサの例では，プログラムカウンタに対して，命令フェッチの後のインクリメントと分岐成立時の書き込みの 2 つのメソッドが同時に呼び出されることがある (図 5の下線部)．

このように 2 つのメソッドが同時に呼び出された場合には，例えば「インクリメントのメソッドの実行をキャンセルして，書き込みのメソッドのみを実行する」というように呼び出されたメソッドを統一的に扱う記述ができる必要がある．

Evelyn では，このような場合の処理は restricted method を用いて記述する．restricted method は，同時に複数のプロセスから呼び出されたときに，それらを統一的に処理するメソッドである．

4.2.1 restricted method 記述例

図 6にプログラムカウンタの記述例を示す．pc は read, inc, write のパブリックなメソッドを持つ．inc と write が同時に呼び出されたときの処理は，プライベートな restricted method，core として記述する．

inc, write メソッドは，呼び出されるとすぐに restricted method を呼び出す．プログラムカウンタの更新という操作は皆 restricted method の内部で行なう．

core が，更新操作を統一的に扱うメソッドである．図中，CALLER とは特別な変数で，当該実行サイクルでその restricted method を呼び出したメソッドのリストが格

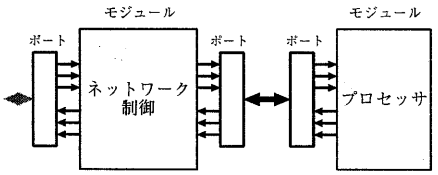

図 7: プロセッサとネットワークの接続

納されている．この例では，インクリメントと書き込みが同時に実行された場合には，書き込みのみを実行して，インクリメントを実行しないように記述している．

for～return 文で，呼び出し側メソッドに対して実行結果を返している．呼び出し側メソッドには次の 3 つの値を返すことができる:

- SUCCESS : 当該サイクルで実行を終えた．呼び出し側メソッドは処理を先に進める．
- CANCEL : restricted method の実行をキャンセルする．呼び出し側メソッドは処理を先に進める．
- RETRY : 次のサイクルでもう 1 度 restricted method を実行する．

4.3 外部モジュールとの通信

今まで説明してきた通信は，命令パイプラインとプログラムカウンタなど，全て内部モジュールへ対して行なう通信であった．

外部のモジュールに対して通信を行なう場合も，内部に対して行なう場合と基本的に同じであるが，1 つ問題がある．それは，外部の通信相手を陽に記述してしまうと設計のポータビリティが大幅に損なわれてしまう，ということである．

そこで，Evelyn では**ポート**というものを用いて外部のモジュールとの接続を表現する．ポートとは，外部のモジュールとの通信を仲介するものである．

外部のモジュールに対してメソッドを呼び出す可能性のあるモジュールには，ポートを用意させる．そして，上位記述でモジュールをインスタンス化する際に，並列するモジュールのポート同士を接続する．

モジュールはメソッドを起動する際に，直接相手のメソッドを起動するのではなく，ポートに対してメソッドを起動する．間接的に相手を指定するので，相手を特定しなくてもプロセッサ，ネットワークを記述することができる．

例として，図 5のプロセッサをネットワークに接続して，並列計算機を構成する場合を考える．

```
module processor{
  port:
    port0; /* ポートの宣言 */
}
```

(a) モジュール部で port0 を宣言

```
void processor::read(){
  y <= port0.read_request( x );
}
```

(b) ポートに対してメソッド呼び出し

図 8: ポートを介して通信

```
connect{
  left:
    the_processor.port0;
  right:
    the_network.port0;
}
```

図 9: ポートの接続

ここでは並列計算機を図 7のように構成する[1]．すなわち，中央にネットワークの制御を行なうモジュールを用意して，それを介してプロセッサを接続する．

まず，図 8(a) のように，プロセッサのモジュール記述でネットワークと通信するためのポート port0 を宣言する．そしてネットワークに対してメソッド呼び出しを行なう場合には，図 8(b) のように，ポートを介して呼び出す．

プロセッサとネットワークそれぞれのポートは，両者をインスタンス化する際に図 9のように接続する．

5 機能合成

本章では，現在構想中の合成ディレクティブの概要を紹介する．

5.1 論理の分割方法の指定

2.1節で述べたように，動作記述を機能モジュールにマッピングする際に，論理を時間的，空間的に分割することがある．

[1]一般にネットワークの制御方法には集中制御方式と分散制御方式が考えられるが，Evelyn で記述する段階ではそれらを区別する必要はない．

時間的分割　Evelynでは，時間的な分割を，signalとsingalの位置関係によって表現する．一連の処理を記述の単位とするので，時間的分割の際に論理を移動する必要はなく，ただ変数の種類を変更すれば済むようになっている．

Evelynでは全命令の時間関係は，signal，singalへのアクセスと依存関係を基に半順序的に順序付けられる．この順序付けからデータフローグラフを作成し，設計者にグラフィカルな情報を提供する．設計者がこの情報から方式，動作レベル設計の見落しを確認できるようにする．

空間的分割　一方，空間的な分割に関しては，合成ディレクティブを用いて指定する．1つの論理合成単位に含めたいパスの集合をディレクティブとして与える．

具体的には，VHDLのentity部のような，論理合成単位の外部インタフェースを指定する記述を与えると，機能合成系がそれに相当するアーキテクチャ部を切り出してくるという形になる．

また，論理合成の手間を考えて，クリティカル・パスを1つの論理合成単位に含めたい場合に設計者をサポートする機構を用意する．

クリティカル・パスとなりうるのは，singalの読み出しから，singalへの書き込みに至るパスである．そこで，そのようなパスのうち論理の深いものを処理系に列挙させる．

列挙されたパスを参考にして，設計者は真にクリティカルなものが1つの合成単位に含まれるように，論理分割のディレクティブを作成する．

5.2　パイプライン実行

Evelynでは，デフォルトでパイプライン実行する論理回路を合成する．

しかしながら，ハードウェアとしてはパイプライン処理することが可能であるが，仕様上パイプライン実行する必要がない，できない場合がある．このような場合のために，合成ディレクティブでパイプライン構成にするかどうか指定することができる．

パイプライン構成にしないように指定すれば，パイプライン同期のための回路は合成されない．

6　おわりに

本稿では，Evelynの設計方針について述べ，動作記述の概要について説明した．

Evelynは，抽象的なモジュール境界の記述から，論理合成/最適化に適した機能モジュールを自動生成することを目標としている．

動作記述部は，一連の動作を記述の単位としており，処理を自然な形で記述できる．また，モジュール間の通信をメソッド呼び出しで表現し，通信を抽象化している．さらに，signalとsingalの2つの変数を用意して，パイプラインの抽象化を行なっている．

謝辞

メンター・グラフィックス・ジャパン株式会社のHigher Education Programの一環として製品とサービスをご提供頂いたことに感謝します．

なお本研究の一部は，文部省科学研究費補助金（基盤研究（C）課題番号09680334，奨励研究（A）課題番号09780268）による．

参考文献

[1] 安浦寛人, 山田輝彦, 他. 特集「ハードウェア記述言語–新しいシステム設計環境の実現に向けて–」. 情報処理 Vol.33, No.11, Nov. 1992, 1992.

[2] 森本貴之, 斉藤一志, 中村宏, 朴泰祐, 中澤喜三郎. 方式レベル記述言語AIDLを用いた高性能プロセッサ設計支援. 情報処理研究会報告 96-ARC-121-8, 96-DA-82-8. pp.57-64., 1996.

[3] 若林一敏, 古林紀哉, 斎藤正一, 加納敏行, 篠原直子, 田中弘人, 中越優佳, 北村晃一. 伝送用LSIを動作合成で開発，機能設計の期間が1/10に短縮. 日経エレクトロニクス 2-12-1996 pp.147-169, 1996.