特別研究報告書

動作レベルハードウェア記述言語 Evelyn の実行モデルとコンパイル手法

指導教官　　富田 眞治 教授

京都大学工学部情報学科

牧田 崇顕

平成 11 年 2 月 12 日

動作レベルハードウェア記述言語 **Evelyn** の実行モデルとコンパイル手法

牧田 崇顕

内容梗概

VHDL や Verilog–HDL といった機能レベルハードウェア記述言語が一般に普及しつつある。しかし、これらの 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、真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。

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のもさることながら、これらの HDL の記述能力自体の低さにも大きな原因があった。多くの HDL では論理合成を強く意識していることもあり、論理合成/ 論理最適化の単位となる機能モジュールに対して動作記述を行うことでハードウェア全体の動作を表現する。

そのため設計者は、設計の最初期から論理合成の結果が物理的な制約を満たすように設計記述を行わなければならない。また設計の初期段階で綿密に記述を行ったとしても、論理最適化の結果物理的制約を満さないということが起こりうる。そのようなときは動作記述をはじめからやり直さなければならない。

これにより、物理的な回路と 1 対 1 に対応するような機能モジュールに対して動作記述を行うという機能レベル HDL の設計アプローチでは、トップダウン設計の実現が非常に困難であることがわかる。

本稿で提案する **Evelyn** では動作モジュールという物理的な境界とは関係ない抽象的なものを単位に動作記述を行い、そこに機能レベルの設計情報を合成ディレクティブとして付加するというアプローチをとる。従って設計の初期には動作設計のみに専心することができる。また物理的制約による変更があっても合成ディレクティブのみを変更すればよく、設計変更が動作記述にまで及ぶことはない。このようなアプローチを採ることで、Evelyn では真のトップダウン設計を実現する。

Evelyn の動作モジュールはモジュール部と behavior から成り、behavior でモジュールの動作の記述を行う。

Evelyn では動作を時間方向に記述し、その記述中の各処理間の時間関係はデータの依存関係によって半順序的に規定される。また順序関係の規定されな

い処理は並行に実行される。時間関係が半順序的なので柔軟な記述が可能となる。さらに、Evelyn では複数サイクルに渡る動作記述ができる。これは代入文に対してその実行に要するサイクル数を指定することによって行う。

Evelyn の behavior のデータ領域はすべて静的に確保され、それぞれのデータ領域が各 behavior 呼び出しと 1 対 1 に対応する。よって動作記述におけるデータの依存関係は、すべて静的に決定することができる。Evelyn の実行は、この依存関係によって得られるデータフロー・グラフのデータの流れである。

このような Evelyn の実行をシミュレーションするため、トランザクションというものを用いる。トランザクションとは、ある変数に対して代入される値と代入に要する時間を表すものである。

時間をサイクル時間で区切り、当該サイクルにおいて実行可能となった代入文があれば、その代入先の変数に対してトランザクションをスケジューリングする。トランザクションのうち時間要素が 0 のものがあれば、値の代入を行う。その代入によって実行可能となった代入文があれば、それに対してもトランザクションのスケジューリングを行う。実行可能な代入がなくなったら、時間を 1 サイクル進める。

これを繰り返し行うことで、Evelyn の並行実行をシミュレーションすることができる。

The Execution Model of a Behavior-Level Hardware Description Language Evelyn and its Compiling Method

Takaaki MAKITA

Abstract

Function–level hardware description languages such as VHDL or Verilog–HDL have come into wide use recently. But by using them, designers can't describe designs more abstruct than they were expected. In short, these function–level HDLs can't realize the true top–down design.

Certainly, one of the reasons of that problem is the incapacity of their processor — specially their optimizer. But the most important reason is the incapacity of these HDL description. Because most of these HDLs are strongly conscious of logic synthesis, designers must describe the behavior of the whole hardware in blocks of function modules which is the unit of logic synthesis and optimize.

So, from the very early stage, designers must consider the physical constrains and describe their designs carefully. Whatever they may do so, the physical constrains will not always be satisfied as a result of logic synthesis. Therefore if not satisfied, designers must re–describe the behavior.

By this reason, with function–level HDLs' approach that designers describe designs in blocks of function modules one–to–one corresponding to physical circuit, it is very difficult to realize the top–down design.

With Evelyn, I propose in this paper, designers describe designs in blocks of the abstruct which don't have relation to the physical border. And function–level information is given by adding to the behavioral description as synthesis directives. Therefore designers can concentrate on describing the hehavior. And even if some modifications caused by the physical constrains are necessary, designers modify only synthesis directives and it is not necessary to modify the behavioral description. Evelyn's approach realize the top–down design method.

Behavior modules consist of the module–part and the *behavior*. In the *behavior*, designer describe the behavior.

With Evelyn, designers describe the behavior in the time order and the operation ordering in the description is partially ordered by the dependence of

each data. Operations which are not ordered are executed concurrently. And designers can describe the behavior as a series of action over steps, by specifing how many cycles it takes to execute the assignment.

the data area of Evelyn's *behavior* is statically allocated. Each area one–to–one corresponds to each behavior call. So all dependences of each data in the behavioral description are statically determined. Execution of Evelyn is the flow of the data along the data–flow graph.

To simulate execution of Evelyn, we use the transaction. the transaction represent the value to be assinged and cycle times need to assign.

Dividing time by the cycle time, if there is an assinment which is able to be executed at the concerned cycle, schedule the transaction on the variable whom the value element of the transaction will be assigned. If there is a transaction whose time element is 0, assign the value. If there is an assinment which is able to be executed by the assignment, schedule the transaction. If there are no assinments which is able to be executed, set the time forward by one cycle.

Repeating this, it is able to simulate Evelyn's concurrent execution.

動作レベルハードウェア記述言語 Evelyn の実行モデルとコンパイル手法

目次

1　はじめに

VHDL や Verilog–HDL といったハードウェア記述言語 (HDL: Hardware Description Language) と、それらからの自動的な論理合成/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。

言語入力は回路図入力に比べて入力や修正が容易になる。これにより設計の効率化が図られてきたが、それで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。

設計の抽象度が上がらないは、HDL の処理系、主に論理最適化系の能力が貧弱であったのもさることながら、これらの HDL の記述能力自体の低さにも大きな原因があった。多くの HDL では、論理合成を強く意識していることもあり、論理合成/ 論理最適化の単位となる機能モジュールに対して動作記述を行うことでハードウェア全体の動作を表現する。そのため、回路が物理的制約を満たすように設計の最初期から論理合成される回路を予測しながら設計記述を行わなくてはならない。これではトップダウン設計は二の次にならざるを得ず、設計の初期コスト、修正コストが大きな負担となってしまう。

VHDL や Verilog–HDL などの HDL は、機能モジュールを単位にハードウェアを記述/論理合成するので、特に機能レベル **HDL** と呼ばれる。機能レベル HDL におけるハードウェアの記述方法、すなわち機能モジュール毎にその動作を記述して個々の動作の集合としてハードウェア全体の動作を実現するという方法では、ゲートの数や遅延などの物理的制約によって記述の変更を余儀なくされる可能性が高く、またその変更コストが非常に大きい。よってトップダウン設計の観点からみて、機能レベル HDL が採ったアプローチは間違いであったと言わざるを得ない。

本稿で提案する動作ベレル HDL“**Evelyn**” では、これらの問題点を解決しトップダウン設計を可能とすることをねらっている。

従来の機能レベル HDL が論理合成/最適化の単位となる機能モジュールに対して記述を行うのに対し、Evelyn では動作モジュールという、物理的な境界とは関係ない抽象的なものを単位に動作記述を行い、そこに機能レベルの設計情報を合成ディレクティブとして付加するというアプローチをとる。こうすれば設計の初期には動作設計のみに専心することができる。また物理的制約による

変更があったとしても合成ディレクティブのみを変更すればよく、設計変更が動作記述にまで及ぶことはない。このようなアプローチを採ることで、Evelynでは真のトップダウン設計を実現することができる。

以下、2章でEvelynの概要、3章でEvelynの動作記述について述べる。4章ではEvelynのコンパイル手法について述べ、5章でまとめる。

2　Evelynの概要

本章では機能レベルHDLの問題点、Evelyn設計にあたっての方針を述べる。

2.1　機能レベルHDLの問題点

VHDLやVerilog–HDLなどの従来のHDLを用いても、当初言われた程にはトップダウン設計にはつながらなかった。これらのHDLは論理合成を強く意識しているため、論理合成/最適化の単位となる機能モジュールに対して記述を行う。

そのため、設計者は1度に最適化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、設計の初期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、トップダウン設計を行うことは二の次にならざるを得ず、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・レベルで誤りが混入する可能性が高い。また、設計の初期段階で綿密にモジュールの分割を行ったとしても、論理最適化の段階でゲートの数や遅延などの物理的制約を満たさないことが判明することはやはり避け難い。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、やはり誤りが混入しやすい。

確かに経験を積んだ設計者ならば物理的制約を勘案し、合成される回路を予想しながら設計を進めることも全く不可能な訳ではない。しかし、論理合成/論理最適化系の癖まで考慮に入れて合成結果を完璧に予測しきることは非常に困難である。従って、論理合成/最適化系からのフィードバックによる修正を完全に避けることは難しい。

以上のことから、機能モジュールを単位に記述を行う機能レベルHDLの設

計アプローチは、トップダウン設計の観点からみて間違いであったといえる。

2.2　理想的なハードウェア設計システム

本節では、真にトップダウン設計を実現する理想的なハードウェア設計システムとはどのような条件を満たすべきかを考察する。

トップダウンな設計工程とは、より抽象度の高い設計を徐々に詳細化していくという過程である。ハードウェア設計における詳細化の工程のうち、HDL がカバーする工程を図 1 に示す。

方式設計後からネットリストを得るまでの過程を支援するのが HDL とその処理系の役目である。すなわち自然言語などで記述された仕様に対して、処理間の並列性や資源共有の可能性を考慮して、データ構造やステージ構成を詳細化し HDL のプログラムの形で記述する。完成した HDL 記述を論理合成/最適化系を利用してネットリストに変換するわけである。

トップダウン設計を実現するためには、このような詳細化の工程を言語レベルでサポートすることが望ましい。すなわち方式設計に近い抽象レベルから実際の論理回路と 1 対 1 に対応する程の抽象レベルまで記述できることが望ましい。

しかし、それだけでは不十分である。高い抽象レベルの記述をサポートすることに加えて、設計変更が容易であることが求められる。何故ならば、ハードウェア設計では下流工程からのフィードバックにより上流である設計変更を迫

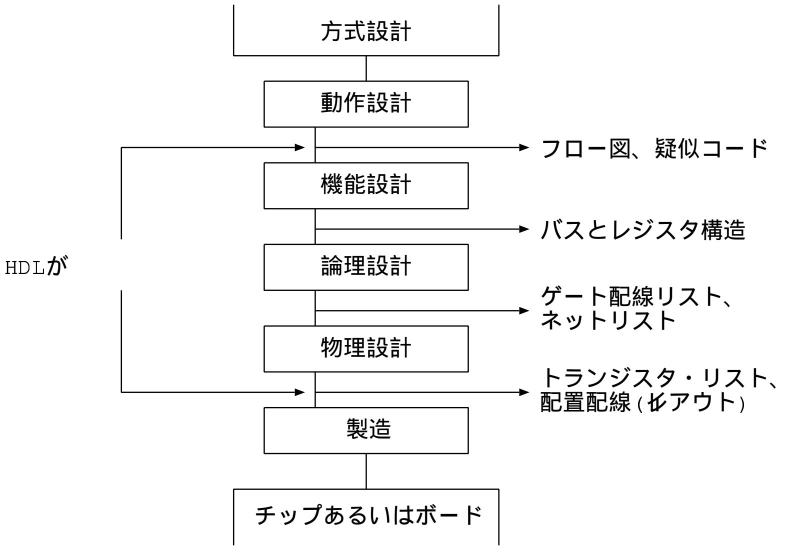

図 1: HDL がカバーする設計工程

られることが多いからである。これはハードウェア設計では物理的制約が非常に強いことに起因する。

アルゴリズムレベルの大規模な修正を行う場合には全て設計し直すことになるのはやむを得ないにしても、例えばパイプライン・ステージの境界を微調整する程度の設計変更が動作レベルの設計まで変更を及ぼすということはできる限り避けたい。そこでハードウェアの設計環境に特に求められるのは、

- 上流工程で混入した誤りをなるべく早い段階で取り除く仕組
- 下流工程での設計変更が上流工程に影響を与えないような仕組

である。設計システムがこれらの仕組を備えればフィードバック・ループを短かく抑えることができ、設計変更のコストを最小限に留めることができる。

ところで既にいくつかの動作レベルHDLとその処理系が、機能レベルHDLの問題点を解決すべく研究されている。しかしそれらはフィードバック・ループの問題を根本的に解決するものではない。

これらの動作レベルHDLの研究は、以下の2つのアプローチを採るものが多い。

1つは機能レベルHDLの欠点をad hocに補おうとするアプローチ、すなわち機能レベルHDLで記述すると記述量が多くなってしまうような特定の回路を簡潔に記述できるようにするというものである。

例えば、Bach [1]では、専用のマクロを提供することでハンドシェイク回路を簡潔に記述できるようにしている。またCyber [2]では、ステート・マシンを簡潔に記述でき、また合成時のオプションにより時分割多重の処理を自動合成することができるようにしている。

もう1つのアプローチは、組み込み用プロセッサなどに記述の対象を限定して、ハードウェアのテンプレートのようなものを与えるものである。

例えばAIDL [3]では、記述の対象をパイプラインプロセッサに絞っている。パイプライン・プロセッサに限定することで、パイプライン・ステージを簡潔に記述できるようにしている。

確かにこれらのアプローチによって、機能レベルHDLに対して記述の抽象レベルが上がることで記述量が少なくなり、結果的に設計コストが削減される。

しかし逆に言えば、これらのHDLが機能レベルHDLに勝るのはこの点に関してのみである。依然として機能モジュールをベースにハードウェアを記述しているため、下流工程からのフィードバック・ループが大規模な設計変更に結

び付きやすいという問題点は何ら解決されていないのである。

また、これらとは別に論理合成を強く意識したSFLというHDLがある。SFLでは、使用できるデータ型は基本的にビット列のみである。また使用することのできる演算も、ビットの切り出し、論理否定、論理and、論理orなどであり、実際の論理回路と1対1に対応するような記述しかできない。用可能なデータ型や演算を制限することにより確かに論理合成は容易になるが、設計の抽象度という点では回路図での設計とほとんど変っていないと言える。

例えばSFLでは加算演算子は機能回路でしか使用できない。機能回路とは論理合成対象としない特別なモジュールである。よって論理合成の対象となるモジュールにおいて加算を行いたい場合などは、adderに相当する回路を記述しなければならない。これでは記述量の増加は避けられない。

そこで、Evelynでは下流工程からのフィードバック・ループを短く抑え、真のトップダウン設計を実現することを目指す。

2.3 トップダウン設計の実現

Evelynの最終的な目標は、真にトップダウンなハードウェア設計環境を実現することである。そのためには、下流工程からのフィードバック・ループを短くすることが是非とも必要である。

2.1節でみてきたように、機能レベルHDLでトップダウン設計を行うことができなかった理由は、実際の論理回路と1対1に対応する機能モジュールを単位として動作記述を行うために下流工程からのフィードバック・ループが長く、それによる設計変更のコストが非常に大きかったためである。

Evelynではフィードバック・ループを短くするために、次のようなアプローチを採る。

1. モジュール境界を抽象化して記述
2. そこに設計情報を付加していくことで詳細化

2.1節で述べた下流工程からの長いフィードバックの問題を解決するためには、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。

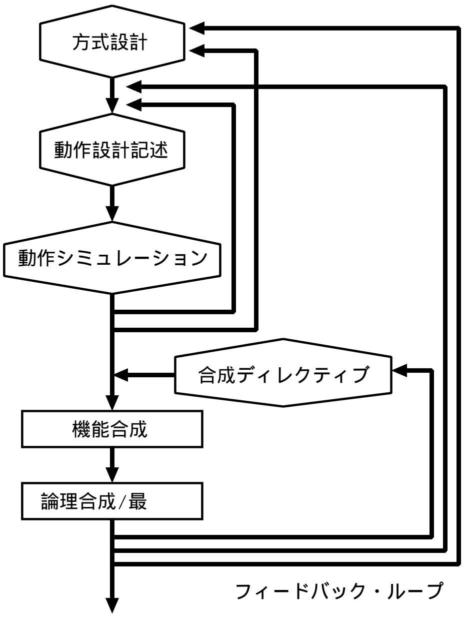

図 2: Evelyn を用いた設計フロー

しかし、単に記述の抽象度を上げただけではそこからの機能合成が困難となる。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際に、抽象的な設計記述に合成ディレクティブと呼ばれる詳細な機能レベルの設計情報を付加するようにする。

詳細な設計情報を合成ディレクティブとして与えることで、記述の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ことができる。

このようなアプローチを採ることで、Evelyn を用いたハードウェアの設計フローは図 2 のようになる。

設計の初期段階では、ハードウェアの動作のみを記述する。この時、機能モジュールのように物理的な境界を表すものに対してではなく動作モジュールと呼ばれる抽象的な塊を単位に記述を行う。ハードウェアを動作単位で記述することで、動作設計記述の中に機能レベルの設計情報が入りこむ必要をなくす。

動作記述に対して動作シミュレーションをし、動作レベルの誤りを取り除く。動作情報のみの簡潔な記述の検証/修正なので効率良く行える。

そして、抽象的な動作記述に後から付加するという形でより詳細な機能レベルの設計情報を記述する。

詳細化された設計記述を機能合成、そして論理合成/最適化する。論理合成/最適化の結果が物理的制約を満たさない場合には上流過程に戻って設計変更す

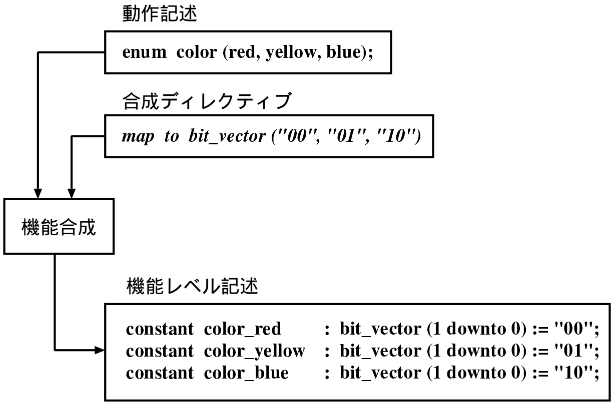

図 3: 列挙型からビット列への変換

ることになるが、多少の変更ならば合成ディレクティブのみを変更すればよく、動作記述にまで手を加える必要はない。

このようにフィードバック・ループを最小限に抑えることでトップダウン設計を実現する。

2.4 Evelyn の言語仕様上の特徴

2.4.1 合成ディレクティブ

Evelyn では、詳細な機能レベルの設計情報を合成ディレクティブとして与える。例えば、一口に記憶回路といっても RAM、flip–flop、latch など様々な実現方法が考えられる。機能合成までには具体的にどのように実現するか、設計情報を与える必要がある。

この合成ディレクティブは、単に合成時のオプションとして与えるのではなく、言語仕様の一部として規定する。これより、ディレクティブのセマンティクスを明確にし、記述の意味が処理系に依存しないようにする。こうすれば記述のポータビリティが高まり、設計者が処理系の癖に悩まされることがなくなる。

2.4.2 抽象データ型のサポート

Evelyn では整数型や実数型の他に、列挙型、多次元配列、構造体や共用体などの抽象データ型をサポートしている。動作記述においてこれらの抽象データ型を用いることによって、複雑なデータを組織化して扱うことができ、設計記述が容易になる。また、これらのデータ型に対し四則演算、関係演算や等値演算などの演算を行うことができ、ソフトウェアのプログラムのように抽象度の高い記述でハードウェアの設計を行うことができる。

これらの抽象データ型は、機能合成時に合成ディレクティブを与えることによって、自動的にビット列に変換される。列挙型での変換例を図 3 に示す。また、演算に関しても必要であれば機能合成時に適当な回路に変換される。このような手法を用いることによって、Evelyn では論理合成可能な回路を抽象度の高い記述で設計することができる。

2.4.3　実行順序

Evelyn では動作記述中の各処理間の時間関係は、データの依存関係によって半順序的に規定される。

時間関係が半順序的なので、時間関係に関して柔軟な記述が可能となる。また多くの動作レベル HDL では、動作記述を行う時点でステージ構成まで決定しなければならないが、Evelyn では純粋に動作だけを記述できる。

以下は Evelyn の動作記述の抜粋である。ここでの $f_j(x_i)$ は、変数 x_iからなる式を表す。

$$
\begin{aligned}
x_1 &= f_0(x_0); && (1)\\
x_2 &= f_1(x_0); && (2)\\
x_3 &= f_2(x_1); && (3)\\
x_4 &= f_3(x_3, x_2); && (4)\\
x_5 &= f_4(x_3); && (5)\\
x_6 &= f_5(x_4); && (6)
\end{aligned}
$$

この記述には図 4 のデータフロー・グラフに示すような依存関係がある。そのためこの記述からは (1) と (3)、(2) と (4) などの間の順序関係のみが規定され、(2) と (3) などの間には順序関係は規定されない。これらの順序関係が規定されなかった文は並行に実行される。

3　Evelyn の動作記述

本章では、Evelyn の具体的な動作記述方法を述べる。

3.1　モジュール

Evelyn ではモジュールを単位にハードウェアを記述する。ここでいうモジュールとは動作モジュールである。すなわち機能モジュールのように物理的な境界と

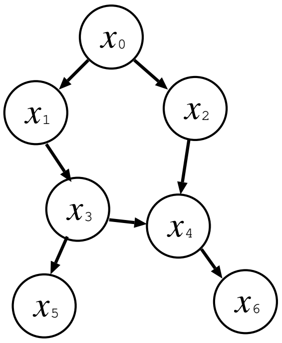

図 4: データフロー・グラフ

は関係ない。機能合成時に動作モジュールを自在に分割/統合して機能モジュールへマッピングすることができるので、動作モジュールは動作を記述しやすいように構成すればよい。

3.1.1 モジュールの構成

モジュールはモジュール部と behavior から成る。モジュール部ではそのモジュールで使用する変数、behavior や内部モジュールの宣言などを行う。behavior はそのモジュールの一連の動作を記述したものである。また 1 つのモジュールが複数個の behavior を持つことが可能である。

以下にモジュール部の構成を示す

```
module identifier {
    generic:
    private:
    public:
};
```

モジュール部は、generic 部、private 部、public 部からなる。これらの役割は以下の通りである。

generic 部 コンパイル時に与えられる変数を宣言する。

private 部 モジュールにローカルな変数および behavior の宣言、また内部モジュールのインスタンス生成を行う。

public 部 モジュールの外部からもアクセスできる変数や behavior の宣言を行う。

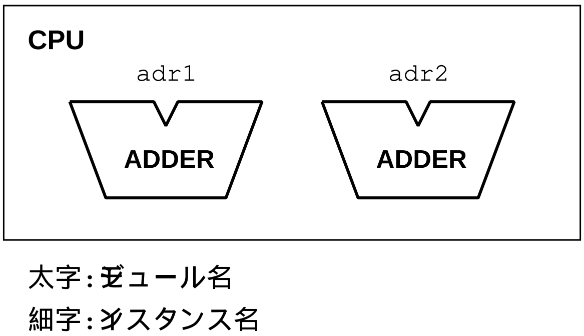

図 5: モジュールと内部モジュール

private 部で宣言された変数や behavior は、そのモジュールに属する behavior、すなわちそのモジュールの private 部または public 部で宣言された behavior からのみ参照または呼び出すことができる。public 部で宣言された変数や behavior は、そのモジュールに属する behavior と、そのモジュールを内部モジュールとして使っている上位モジュールに属する behavior から参照または呼び出すことができる。

3.1.2　モジュールの階層化

モジュールは、その定義の中で他のモジュールを使う (すなわち、階層表現をする) ことができる。階層表現をするとき、下位のモジュールの 1 個 1 個 (インスタンス) を内部モジュールと呼ぶ。たとえば図 5 のような構成のとき「CPU というモジュール (部品種) は ADDER というモジュール (部品種) を内部モジュール (部品) として 2 個使っていて、その名前 (インスタンス名) は adr1、adr2 である。」という。

この内部モジュールのインスタンス化の記述例を以下に示す。

```
module CPU {
  ······
  private: ADDER   adr1, adr2;
  ······
};
module adder { ······ };
```

モジュール CPU のモジュール部で、モジュール ADDER のインスタンス化を

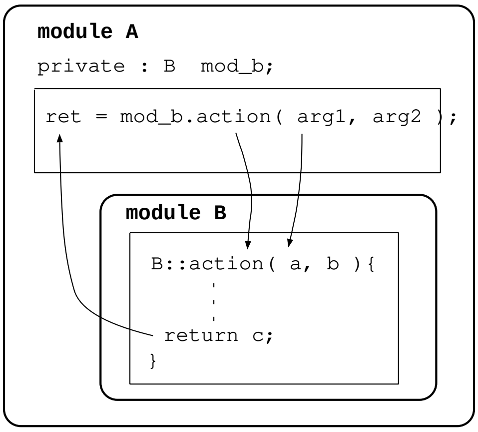

図 6: モジュール間の通信

宣言している。そのインスタンス名がそれぞれ adr1、adr2 であることを記述している。

3.1.3 モジュール間の通信

Evelyn ではモジュール間の通信を behavior の呼び出しに抽象化している。behavior 呼び出しの際の引数としてデータを相手モジュールに渡し、その返り値として相手モジュールからデータを受け取る。

例えば、モジュール A がモジュール B を内部モジュールとして使っているとする。モジュール A の behavior からモジュール B の behavior を呼び出した場合には、その引数がデータとしてモジュール A からモジュール B に渡される。モジュール B では、その呼び出された behavior に記述された処理が行われ、値が返される。そのデータが今度はモジュール B からモジュール A に渡される(図 6)。

先に述べたようにモジュールは複数の behavior を持つことができ、呼び出し側ではそれらの behavior の名前を指定して呼び出しを行う。モジュールの行う動作をそれぞれ 1 つの behavior にまとめて記述し、呼び出し側ではそのモジュールに行わせたい処理に対応した behavior を呼び出す。このような方法を用いることで、モジュールのインターフェースに関する記述を減らすことができる。つまり制御信号に関する記述をする必要がなくなり、データ信号に関する記述のみでよくなる。

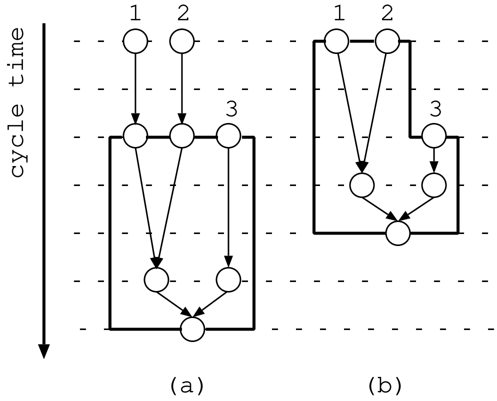

図 7: 引数の時間関係

3.2 behavior

3.2.1 behavior の呼び出し

behavior は複数の引数を持つことができるが、これらの引数は behavior の呼び出しの際にすべてが揃っている必要はない。

2.4.3 節で述べたように、Evelyn ではデータフロー・グラフで依存関係が規定されていない処理は並行に実行される。よって順序関係が規定されていないノードに対応する変数を引数とした場合には、それぞれの引数に有効なデータが代入されるのには時間差が生じる。これらの引数の中で、他の引数に比べて有効なデータが代入されるのが遅れるものがあったとしても、先に有効なデータが代入された引数から呼び出し先のモジュールにデータを渡してやることで、遅れているデータを用いない処理から先に進めていくことができる。

例として、第 1 引数と第 2 引数の乗算を計算し、さらにその結果と第 3 引数の加算を行う behavior を呼び出す場合を考える。この behavior では乗算を 3 サイクルで実行し加算を 1 サイクルで実行するとする。一方で呼び出す側では、第 1 引数と第 2 引数に比べて第 3 引数の値が有効になるのが 2 サイクル遅れるとする。図 7 はその引数と時間の関係を表したものである。図での 1、2、3 はそれぞれ第 1、第 2、第 3 引数を表している。図の (a) は、引数がすべて揃ってから behavior を呼び出した場合の時間関係で、(b) は Evelyn での呼び出しの場合である。behavior の呼び出し時に引数がすべて揃っていなければならないとすると、第 1、第 2 引数が揃ってから処理が終了するまでに合計 6 サイクルかかる。呼び出しの前では 1 と 2 がデータ 3 を待ち、behavior ではデータ 3 が乗算の結果を待つことになり実行効率が悪い。しかし、Evelyn で用いた方式なら

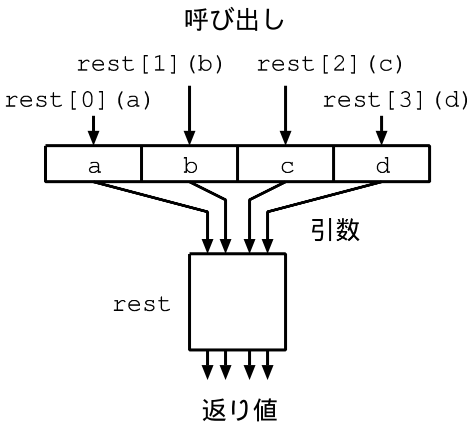

図 8: 制約 behavior

ば 4 サイクルで処理が終了する。第 3 引数がそろっていなくても、乗算の実行を先に行うことができるからである。この方式だと待ち時間が少なくてすむ。

3.2.2 制約 behavior

behavior の呼び出しでハードウェアをモデリングする場合、同一の behavior が同時に呼び出された場合の処理が問題となる。すなわち、一般のソフトウェアのプログラムならばこのような場合はそれらが排他的に実行されればよいことが普通であるが、ハードウェアの場合にはそれらをまとめて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制約 behavior を用いて記述する。

制約 behavior の宣言は以下のように行う。

int rest[4] (**int** arg);

配列型の宣言はその behavior が制約 behavior であることを表す。つまり rest は要素数が 4 で int 型を返す制約 behavior であることの宣言である。つまり rest は 4 箇所から呼び出すことができる制約 behavior である。

制約 behavior は前節で述べた配列型の behavior の特殊なものである。配列型の behavior が各呼び出しに対してそれぞれ独立に動作するのに対し、制約 behavior では各呼び出しを統一的に扱う動作が 1 つ存在するものである。図 8 は上記の rest を図で表したものである。

呼び出しを統一的に扱うということは、その behavior へ渡されたデータを統

一的に扱うということである。よって制約 behavior は、すべての呼び出しの引数を参照できる。さらにその値が有効な値であるかを調べることができる。例えば、図 8 の制約 behavior の rest の宣言は引数が 1 つでそのデータ型は int 型であるが、behavior の動作記述の部分ではそれは int 型の配列となる。図 8 のような呼び出しをされた場合、arg[0]、arg[1]、arg[2]、arg[3] の値はそれぞれ a、b、c、d の値となる。

3.2.3 active behavior

behavior の中には他のモジュールから呼出されることで動作を始めるものと、他のモジュールから呼び出されなくても実行の開始とともに動作を始めるものがある。後者のものを active behavior と言い、システムを始動させる役割を持つ。

active behavior は何か特別な宣言をすることによって記述するのではなく、引数を一つも持たない behavior が active behavior となる。

3.3 実行サイクル数の指定

Evelyn では、複数サイクルに渡る動作を記述することができる。各代入文に対してその実行に要するサイクル数を指定することによって、複数サイクルに渡る動作を記述する。

2.4.3 節での動作記述を以下のように書き換える。

$$x_1 = f_0(x_0) \quad after\ 1\ cycle; \tag{1}$$
$$x_2 = f_1(x_0) \quad after\ 2\ cycle; \tag{2}$$
$$x_3 = f_2(x_1) \quad after\ 2\ cycle; \tag{3}$$
$$x_4 = f_3(x_3, x_2) \quad after\ 3\ cycle; \tag{4}$$
$$x_5 = f_4(x_3) \quad after\ 1\ cycle; \tag{5}$$
$$x_6 = f_5(x_4) \quad after\ 2\ cycle; \tag{6}$$

これは (1)、(5) は実行の開始から完了までに 1 サイクル要すことを表している。(2)、(3) と (6) は 2 サイクル、(4) は 3 サイクルをそれぞれ実行に要する。この実行過程をデータフロー・グラフで表したものを図 9 に示す。

時刻 $T = t_0$ に上記の動作記述の実行を開始する場合を考える。変数 x_0 はどの変数にも依存していないから、時刻 $T = t_0$ に (1) と (2) の実行を開始する。

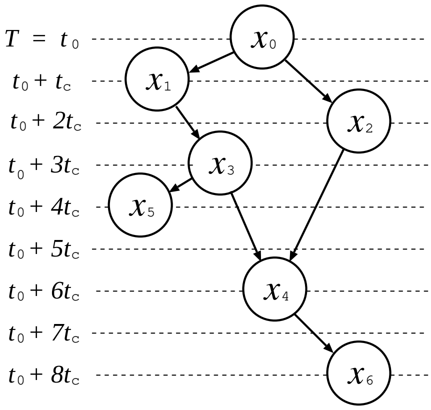

図 9: データの流れと時間の関係

サイクル時間を t_c とすると、時刻 $T = t_0 + t_c$ に (1) の実行が完了する。これにより x_1 が得られたので、(3) の実行を開始する。この時、まだ (2) は実行中である。

時刻 $T = t_0 + 2t_c$ に (2) の実行が完了する。(3) は実行中である。(2) の実行が完了していても、データフロー・グラフから (4) の実行開始は (3) の実行完了後でなければならないから、まだ (4) の実行は開始されない。

時刻 $T = t_0 + 3t_c$ に (3) の実行が完了する。ここで (4) の実行が開始される。また (5) の実行も開始される。

以降同様に実行され、時刻 $T = t_0 + 8t_c$ に (6) の実行が完了する。

また代入文において $after$ 以降を省略した場合は、 $after\ 0\ cycle$ と記述したのと同じであるとする。

4　コンパイル手法

本章では、Evelyn で記述されたハードウェアをシミュレーションするためのコンパイル手法について述べる。

4.1　動作シミュレーション

ハードウェアの設計では、設計プロセスの下流になるほどそのシミュレーションで扱う情報が詳細になり、シミュレーションに時間がかかる。よって、設計プロセスのできるだけ早い段階である上流で設計の誤りを検出するようにしなければならない。

そこで Evelyn では、合成ディレクティブを与える前の動作記述に対して動作

シミュレーションを行うようにしている。この段階のシミュレーションは、回路のタイミングなど詳細な情報を考慮する必要がないので、低コストなシミュレーションを行うことができる。また、シミュレーションの結果その動作記述に誤りがあったとしても、そこに機能レベルの記述がない分、修正は容易になると思われる。

Evelyn の動作記述では、代入文に対して実行に要する時間を指定する。この

シミュレーションにおいて重要となるのは、どの時間にどの文が実行され、変数の値がどのように変化するかである。

Evelyn ではデータに依存関係のない代入文は並行に実行されるとしている。シミュレータでは時間をサイクル時間ごとに区切り、当該サイクルにおいて実行されるべき演算等をすべて行い、時間を 1 サイクル進めると同時にそれらの結果を代入先の変数へ代入することによって並行実行をシミュレーションする。

一方でシミュレータではこれらの代入文の実行は逐次実行であり、サイクル時間で区切られた間の部分ではどのような過程で実行が行われているかを知りたい場合がある。例えば、すべての代入文において $after$ 以降を省略した場合などである。このような場合にサイクル時間のみを時間の単位としてシミュレーションを行うと、最終的な結果のみしか知ることができないのである。

よってこれらの逐次実行をシミュレーションするために、 δ 時間というものを用いる。 δ 時間は 1 サイクル時間に比べて非常に短い時間であり、何回 δ 時間を足しても 1 サイクル時間に比べ無視できるものとする。そしてシミュレーションにおいては、代入文を実行するのに δ 時間かかるとするのである。こうすることにより、どの代入文を実行すると変数の値がどのように変化するかを知ることができるようになる。

4.2　コンパイル

4.2.1　実行の単位

データフロー・グラフにおけるノード間の関係は、代入文における左辺の変数と右辺の変数の関係であるとも言える。つまり Evelyn では代入文が実行の単位となる。データフロー・グラフにおける Evelyn の実行過程とは、あるノードが依存しているすべてのノードに有効なデータが揃ったらそのノードにデータを送るという過程である。これは、代入文においてすべての右辺の変数の値が有効になったら右辺の式を計算し左辺の変数へ代入するということである。

Evelyn の代入文が持っている情報としては以下のものがある。

- 右辺の変数名。すなわちその代入先変数が依存している変数名。
- 左辺の変数名。
- 右辺の式。スケジューリングの対象となる動作。
- その代入の実行に要するサイクル時間。

コンパイラはこれらの情報を各代入文ごとにまとめて、シミュレータに渡す。この情報に基づいてシミュレータは Evelyn の実行シミュレーションを行う。

4.2.2 変数のリネーム

Evelyn の動作記述の実行順序は、2.4.3 節で述べたようにデータの依存関係を表すデータフロー・グラフによって決定される。しかし、記述の仕方によってはこのような方法をそのまま用いることができない場合が存在する。

例えば、

$$
\begin{aligned}
x_1 &= f_0(x_0); \qquad (1)\\
x_2 &= f_1(x_1); \qquad (2)\\
x_1 &= f_2(x_2); \qquad (3)\\
x_3 &= f_3(x_1); \qquad (4)
\end{aligned}
$$

のような記述を行った場合、そのデータフロー・グラフに閉路ができる。データフロー・グラフに閉路ができてしまうと、データの依存関係が決定できなくなる。つまり (4) における x_1 が (3) を実行する前のものなのか、実行した後のものなのかが決定できないのである。

このような場合の処理として、変数のリネームを行う。データフロー・グラフに閉路ができてしまう代入文がある場合には、その代入先の変数の名前をかえることでデータフロー・グラフの閉路をなくすことができる (図 10)。

$$
\begin{aligned}
x_1 &= f_0(x_0); \qquad (1)\\
x_2 &= f_1(x_1); \qquad (2)\\
x_1' &= f_2(x_2); \qquad (3)\\
x_3 &= f_3(x_1'); \qquad (4)
\end{aligned}
$$

データフロー・グラフに閉路ができる条件は、左辺の変数に対応するノード

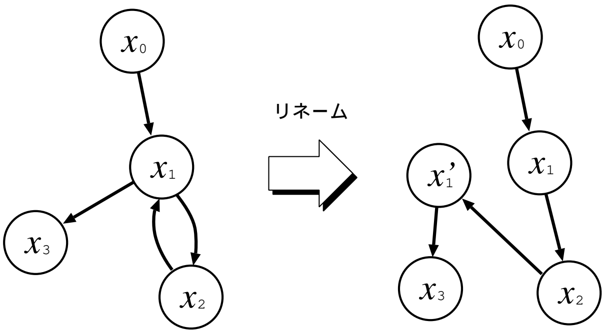

図 10: リネームによる閉路の除去

から右辺の変数に対応するノードのうちの少なくとも 1 つのノードへのパスが存在することである。コンパイル時にこのような代入があった場合には、左辺の変数の名前を変える。さらにそれ以降の文においてリネームの対象となった変数を参照しているものがあれば、その名前も変換する。

4.2.3　behavior の扱い

Evelyn の behavior は一般のプログラミング言語のように、呼び出し時に動的にデータ領域が確保されるようなものではない。データ領域は全て静的に確保される。この意味で Evelyn の behavior はむしろ VHDL のコンポーネントに近いものである。つまり behavior の呼び出しとは、それによって新たにデータ領域が確保されるものではなく、常に存在している 1 つの behavior 領域に対してデータを渡すことで実現される。よって同一の behavior が複数の箇所から呼ばれる場合には、その呼び出しの数に相当するだけの領域をあらかじめ用意しておかなければならない。

こうして確保されたデータ領域と、その behavior の呼び出しとは 1 対 1 に対応するものである。よって、それぞれの呼び出しにおいて、実引数と仮引数の対応や、その返り値がどの代入文に代入されるかはすべて静的に決められる。つまりこれは、behavior の呼び出しはその behavior をすべて展開することでデータフロー・グラフを作成することができることを意味する。

4.3　シミュレーション手法

4.3.1　トランザクション

トランザクションとは、代入に要する時間、代入される値を表したものである。以下ではそれを (“値要素”, “時間要素”) で表し、第 1 項で代入される値、

第 2 項であと何サイクルで値が代入されるのかを表す。

時間要素が 0 であれば、その値は現時刻の値である。時間要素が 0 でないならば、その値は変数の将来値である。

値が一定時間後に、代入先変数に代入されるようにスケジューリングされたとき、トランザクションが代入先の変数に対してスケジューリングされたという。例えば、変数 x に対してトランザクション (4, 3) がスケジューリングされた場合、スケジューリングされてから 3 サイクル後に x の値が 4 になることを意味する。

変数 x の値が ν だとすると、

$$y = x \quad after\ \tau\ cycle;$$

という代入文は、変数 y に対してトランザクション (ν, τ) をスケジューリングするということである。

4.3.2 スケジューリング

δ 時間を用いた実行スケジューリングの手順を以下に示す。ここで T は時刻、S はその時刻にスケジューリングされたトランザクションの数を表す。R はその時刻にスケジューリングされたまま代入されないで残っているトランザクションの数、すなわち時間要素が 1 以上のトランザクションの数を表す。

また各変数には、その値が有効かどうかを表すフラグを持たせる。これが 'valid' の時値は有効、'invalid' の時は無効とする。代入文にもそれが既に実行済かどうかを表すフラグを持たせる。これが 'finished' のとき実行済であり、'unfinished' の時はまだ実行されていない代入文であることを示す。

1. T、S、R に 0 を代入する。以下では $T \leftarrow 0$ のような形式を用いる。
2. 最初の代入文に関するデータを読み込む。(ここでいう最初とは、あくまでコンパイラから渡されたデータの順序でのことであり、実際のプログラムでの記述の順序とは必ずしも一致するものではない。)
3. その代入文のフラグが 'finished' であれば 5. へ。
   また右辺の変数のフラグのうち、1 つでも 'invalid' のものがあれば 5. へ。
4. 右辺の式の計算を行う。その結果を V 、またその代入文の実行に要するサイクル時間を T とすると、その代入先の変数に対してトランザクション

(V,T) をスケジューリングし、$S \leftarrow S+1$ 、$R \leftarrow R+1$ を実行する。
またその代入文のフラグを ‘finished’ とする。

5. 現在対象としている代入文がコンパイラからの最後の入力であれば 7. へ。
6. 次の代入文に関するデータを読み込み、2. へ。
7. S の値が 0 かどうかを調べる。もし 0 であれば 9. へ。
8. スケジューリングされたトランザクションのうち、時間要素が 0 のものに対して、そのトランザクションの値要素を代入先の変数へ代入する。このときに $T \leftarrow T+\delta$ 、$R \leftarrow R-1$ を実行し、代入先変数のフラグを ‘valid’ とする。
   これを時間要素が 0 であるすべてのトランザクションに対して行う。その後、$S \leftarrow 0$ を実行し 3. へ。
9. R の値が 0 かどうかを調べる。0 でなければ $T \leftarrow T+T_c$ を実行し、8. へ。T_c はサイクル時間を表す。このときすべてのトランザクションの時間要素が 1 減らされる。
   Rが 0 であれば終了。

4.3.3　実行例

以下の動作記述を上で述べた方法を用いてスケジューリングを行った場合に、その動作が正しくシミュレーションできることを示す。

$$\begin{array}{lll} a = 1; & & (1) \\ b = 2; & & (2) \\ c = 3; & & (3) \\ d = a * c & after\ 2\ cycle; & (4) \\ e = b + c & after\ 1\ cycle; & (5) \\ f = e - d & after\ 1\ cycle; & (6) \end{array}$$

時刻 $T = 0$ において右辺の値がすべて有効なのは (1)、(2)、(3) であるから a、b、c に対してそれぞれトランザクション $(1,0)$、$(2,0)$、$(3,0)$ がスケジューリングされる。

ここでスケジューリングされたトランザクションの時間要素はすべて 0 であるから、まず a への代入が行われ、$T = \delta$ となり、以下同様にして b、c への代入が行われる。

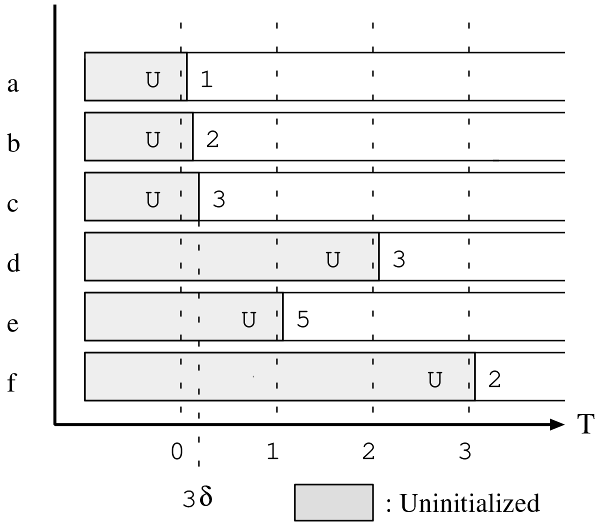

図 11: 各変数の値の変化

$T = 3\delta$ の時点で (4) と (5) の右辺の変数がすべて有効になり、d に対してトランザクション (3,2) 、e に対してトランザクション (5,1) がスケジューリングされる。スケジューリングされたトランザクションに時間要素が 0 のものはないので、当該サイクルにおいて実行される代入はなくなる。

ここで時間を 1 サイクル時間進め $T = 1$ となる。この時点でのトランザクションは、d に対して (3,1) 、e に対して (5,0) がスケジューリングされている。よって e に対して 5 が代入される。(6) ではまだ d の値が有効ではないので、スケジューリングは行われない。

$T = 2$ に d に対するトランザクションの時間要素が 0 となり、d に 3 が代入される。そして fに対して (2,1) がスケジューリングされる。

$T = 3$ に f に 2 が代入される。

この実行における変数の値の変化を図 11 に示す。

5 おわりに

本稿では Evelyn の設計方針と実行モデルについて述べた。

Evelyn は真のトップダウン設計をめざしたハードウェア記述言語である。そこで従来の機能レベルハードウェア記述言語ではなぜトップダウン設計を行うことが困難であったのかについての考察を行った。これらの機能レベルハードウェア記述言語は、物理的な境界に対応する機能モジュールというものを単位に記述を行うというアプローチをとるため、トップダウン設計が困難であるこ

とが分かった。そこでEvelynでは、動作モジュールを単位に記述を行うというアプローチをとる。動作モジュールは機能モジュールとは関係のない抽象的なものである。ハードウェアを動作単位で記述することで、動作設計記述の中に機能レベルの情報が入らないようにする。この動作記述に後から詳細な機能レベルの情報を付加するというアプローチをとることで、下流工程からのフィードバック・ループを最小限に抑えることができる。これによりEvelynでは真のトップダウン設計を実現している。

この動作モジュールはモジュール部とbehaviorから成り、モジュール間の通信は相手モジュールのbehaviorを呼び出すことによって行われる。これによってモジュール間の通信に関する記述を簡潔に行うことができる。また、Evelynのbahaviorは引数がすべて揃っていなくても呼び出すことができるという特徴がある。これによって引数の中にデータが遅れるものがあったとしても、そのデータを用いない処理から先に進めていくことができ、モジュール間の通信で無駄なデータの待ち時間をなくすことができる。

また本稿ではEvelynで記述されたハードウェアをシミュレーションするためのコンパイル手法について述べた。

各代入文を実行の単位とし、コンパイラは各代入文のデータ依存関係を調べて、それをシミュレータへ渡す。そのデータに基づいてシミュレーションが行われる。このシミュレーションの際の実行スケジューリングを行うために、本稿ではトランザクションというものを用いたシミュレーション手法について述べた。時間をサイクル時間で区切り、実行可能となった代入文代入先の変数に対してトランザクションをスケジューリングするという方法を用いることで、Evelynの並行実行をシミュレーションすることができることを示した。

謝辞

本研究の機会を与えて下さり、適切な御指導を賜わった富田眞治教授に深甚な謝意を表します。

また、貴重な御助言をいただいた森眞一郎助教授、五島正裕助手に深く感謝致します。

最後に、日頃から様々な角度からご助力頂いた、京都大学大学院情報学研究科通信情報システム専攻富田研究室の諸兄に心より感謝致します。

参考文献

[1] 西田浩一, 山田晃久, 神戸尚志,Kay, A., 野村俊夫: ハードウェアコンパイラBach, 信学技報 CPSY97-87, pp. 143-148 (1997).

[2] 若林一敏, 古林紀哉 他: 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, No. 655, pp. 147-169 (1996).

[3] 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, 情処技報 96-ARC-121-8, 96-DA-82-8l, pp. 57-64 (1996).

[4] 舟本一久: 動作レベルハードウェア記述言語 Evelyn, 修士論文, 京都大学大学院工学研究科情報工学専攻 (1998).

[5] Zainalabedin Navabi (佐藤一幸 訳) : VHDL の基礎, 日経 BP 出版センター (1994) .

[6] Douglas L. Perry (メンター・グラフィックス・ジャパン 訳, 今井正治/山田昭彦 監訳) : VHDL, アスキー出版局 (1996).