計算機アーキテクチャ 115－8
設 計 自 動 化 78－8
（1995. 12. 14）

デルタ遅延に基づくVHDL記述のスライシングアルゴリズム

一ノ瀬 茂† 野村雅也‡ 岩井原瑞穂‡ 安浦寛人‡

†九州大学 工学部 情報工学科 ‡九州大学 大学院 総合理工学研究科 情報システム学専攻

〒816 福岡県春日市春日公園 6-1

E-mail: {*ichinose,mnomura,iwaihara,yasuura*}*@is.kyushu-u.ac.jp*

我々はLSI設計の検証を支援する手法として，プログラムの大規模依存解析手法であるプログラムスライシング技術をハードウェア記述言語に適用することを提案しており，そのハードウェア記述言語の1つであるVHDLに対するスライシングの研究を行っている．本稿ではデルタ遅延モデルに基づいたVHDL記述を対象としたスライシングアルゴリズムを示す．VHDL記述中に存在する依存関係を制御依存，データ依存，信号依存に分類してグラフとして表現し，その上での探索問題としてスライシングを定式化している．

和文キーワード: プログラムスライシング，VHDL，デルタ遅延，依存解析，グラフ表現，設計検証

Slicing Algorithm for Delta-Delay VHDL Descriptions

Shigeru Ichinose†, Masaya Nomura‡, Mizuho Iwaihara‡and Hiroto Yasuura‡

†Department of Computer Science and Communication Engineering, Fuculty of Engineering, Kyushu University

‡Department of Information Systems, Iterdisciplinary Graduate School of Engineering Sciences, Kyushu University

6-1 Kasuga-koen, Kasuga-shi, Fukuoka, 816 Japan

E-mail: {*ichinose,mnomura,iwaihara,yasuura*}*@is.kyushu-u.ac.jp*

We have proposed applying programming slicing to hardware description language; program slicing is a method of analyzing and modifying behavior of large scale programs. We choose VHDL as a target language, and discuss slicing problems of VHDL descriptions, where VHDL is restricted to zero-time delay, called delta-delay VHDL. We classify dependence within descriptions into control, data, and signal dependences, and define a graph representation of those dependences. Slicing algorithm are defined as searching problems on this graph representation.

keywords: program slicing, VHDL, delta-delay, dependence analysis, graphic representation, design verification

1 はじめに

集積回路技術の進歩により大規模な回路を比較的安価にかつ短期間で製造できることが可能となり，さらにその設計において，ハードウェア記述言語 (HDL: Hardware Description Language) からの論理合成の自動化が可能になった．今後，ハードウェア記述言語による設計資産の蓄積・巨大化が予想され，その設計対象となるハードウェア自体も大規模化・複雑化していくことが予想される．ハードウェア記述の設計検証，保守，再利用の問題が重要さを増すなか，その手法の様々な研究が盛んになされている．

現在，我々は HDL によって設計された LSI 回路の設計検証を支援する手法として，ソフトウェア工学の分野で研究されているプログラムスライシング技術を HDL 記述に適用することを提案しており，標準的 HDL の 1 つである VHDL (VHSIC HDL: Very High Speed Integrated Circuit HDL) に対するスライシングの研究を進めている [1].

プログラムスライシング技術とはプログラム中の依存関係を解析し，注目する動作に影響を与える部分を抽出する技術であり，今日では，検証，デバッグ，保守へと広範囲に応用されている [4].

VHDL 記述にスライシングを適用することで，記述中の注目した文に依存関係を持つ全ての文を抽出することができる．スライシングの応用として，ソフトウェア分野と同様の応用が期待でき，デバッグ，テストパターンの生成，保守，設計再利用，形式的検証などへの応用が考えられる．

本稿では信号の遅延時間をゼロとする，いわゆる VHDL デルタ遅延モデル [2] に対象を限定し議論する．抽象度の高いシステムレベルの記述を対象とする場合は，実時間遅延の記述はわずらわしくデルタ遅延モデルで十分な場合が多い．本稿では VHDL 記述を記述内の依存関係を弧によって分類した有向グラフとして表現し，その上での探索問題としてスライシングを定式化する．

2節でプログラムスライシングについて簡単に説明し，3節で VHDL 記述にスライシングを適用する際の基本的考察．4節では VHDL 記述の制御の流れと依存関係を表す制御フローグラフ，definition-use グラフ，プロセス依存グラフをデルタ遅延モデルに基づいて定義し，5節でそのグラフ上でスライスを求めるアルゴリズムを説明し，6節で考察を述べまとめる．

2 プログラムスライシング

プログラムスライシング (以降，単にスライシング) とはプログラム中の依存関係を明らかにする技術であり，Mark Weiser によって 1982 年に考案されたものである [3]．プログラム中のある文に対してスライシングを行うとは，その注目した文に依存関係を持つ文を抽出することであり，その抽出された文の集合をスライスと呼ぶ．

2.1 依存関係

スライシングの際に解析する依存関係には大きく分けて次の 2 つが提案されている．

- 制御依存
  文 s_1 から文 s_2 への制御依存があるとは，s_1 は WHILE 文か IF 文であり，文 s_2 の実行の有無が文 s_1 の実行結果に直接依存する場合．
- データ依存
  文 s_1 から文 s_2 へのデータ依存があるとは，文 s_1 におけるある変数 v の定義が v を使用している文 s_2 に到達する場合．

データ依存における到達するとは以下の場合をいう．

1. v は文 s_1 で定義される．かつ，
2. v は文 s_2 で参照される可能性がある．かつ，
3. s_1 から s_2 への制御パスが存在し，そのパス中で v が更新されない．

2.2 スライス

前節の様な依存関係を解析して得られるスライスは注目した文に関する動作が保存されている．つまり，このスライスとは実行可能な部分プログラムであるといえる．このスライスにも大きく次の 2 つが提案されている．

- スタティックスライス (*static slice*)
  ある文の実行に影響を与える可能性のある文の集合
- ダイナミックスライス (*dynamic slice*)
  ある入力に対する実行により実際に影響を与えた文の集合

スタティックスライスはプログラム中のある機能を実現している部分だけを取り出すことができるため，ソフトウェアの改造，統合，リバースエンジニアリング，モジュール強度を調べるメトリックスの研究などに応用されている．ダイナミックスライスはある入力を与えてプログラムを実行した場合に，ある変数の値に影響を与えた部分を取り出すことができるため，エラーの原因を調べるデバッグや改造によって影響を受ける部分のテストなどに応用されている．

2.3 スライシングの例

スライシングの例として文献 [4] より，図 1のプログラムを使用する．このプログラムは文字列からなるテキストファイルの行数 nl，語数 nw，文字数 nc をカウントするプログラム WordCounter である．

```
1   InWord:=NO;
2   nl:=0;
3   nw:=0;
4   nc:=0;
5   get(c);
6   while c≠EOF loop
7      nc:=nc+1;
8      if c=CR then
9         nl:=nl+1;
       end if;
10     if c=' ' or c=CR or c=TAB then
11        InWord:=NO;
       else
12        if InWord=NO then
13           InWord:=YES;
14           nw:=nw+1;
          end if;
       end if;
15     get(c);
    end loop
16  put(nl);
17  put(nw);
18  put(nc);
```

図 1: プログラム WordCounter

図1のプログラムの文字数ncを出力している命令18に関するスタティックスライスを求めてみると図2のようになる．この図に明らかなように，18行あったプログラムが6行になっている．出力命令18の実行に影響を与える可能性のある命令，すなわち，文字数ncの計算に関係する部分だけが抽出された，実行可能な部分プログラムが得られている．

```
4   nc:=0;
5   get(c);
6   while c≠EOF loop
7      nc:=nc+1;
15     get(c);
    end loop;
18  put(nc);
```

図 2: WordCounter の命令18に関するスタティックスライス

3 VHDL記述のスライシング

3.1 ソフトウェア言語との違い

VHDL記述にスライシングを適用する際，ソフトウェア言語とは記述の対象が異なるためソフトウェア言語に対する手法がそのまま適用できるわけではない．それらの違いの1つとして並列性が挙げられる．実際のディジタル・システムでは，システムを構成する各要素は全て同時に動作しており，仕事は並列に行われている．このようなシステムを記述するVHDLにおいて，その記述は転送文，コンポーネント記述，ゲートあるいは論理ユニットが全て同時に実行されたかのように実行される．ソフトウェア言語にもAdaやOccum2といった並列言語があるが，ソフトウェア並列言語での同じ入力を与えた2つの異なる実行は，プロセスの進度が予測できない上に，プロセス間での通信の相手が非決定的に選択される記述があるため，それらは異なる動作と実行履歴を生み出すであろう．それに対してVHDLはシミュレーション言語であり，同じ入力を与えればその振る舞いも必ず同じものとなる決定性のセマンティクスが与えられている．ソフトウェア並列言語は非決定性であるが，VHDLは決定性であるということより，ソフトウェア並列言語では解析できないような代入などのタイミングも解析できるといえ，このタイミングを解析できればより精度の良い(記述量の小さい)スライスを求めることができると考えられる．

3.2 VHDLの基本用語

以下の議論を進めていく上で使用されるいくつかのVHDLの用語について説明する．

イベント(*event*)とトランザクション(*transaction*)は以降で論じる信号代入でしばしば言及される．波形が代入先の信号の値に変化を与える時，イベントが代入先の信号に生じたという．値がある時間後に代入先の信号に代入されるようにスケジューリング(*scheduling*)された時，トランザクションが代入先の信号のドライバ(*driver*)に置かれたという．また，トランザクションによってドライバが新しい値を獲得する時，新しい値が前の値と異なっているかどうかにかかわらず，そのドライバは活性状態(*active*)であるという．

3.3 VHDLのデルタ遅延モデル

VHDLがソフトウェア並列言語より複雑にしているものとして，AFTER節などによって記述される実時間遅延がある．本稿ではこの実時間遅延を扱うことを避け，デルタ遅延VHDLと呼ばれるゼロ時間遅延のみに制限されたVHDL(以下単に，デルタVHDL)のモデルを文献[2]より取り上げて，以下のような制限のもとで議論を進める．

- アサーション文，手続き，手続き呼び出し，case文，next文，exit文，そしてreturn文を除外する．
- 明示的な待機文は扱わず，それぞれのプロセスには非明示的な'wait for 0*ns*'という文があるものとする．

デルタVHDLでは図3のようなプロセスモデルをとっている．プロセスは順次文だけからなり，その評価はゼロ時間で終了する．プロセスの最も簡単な形はアーキテクチャ文部の信号代入である．VHDLは入力に対して各プロセス毎に文を順次的(sequential)に評価してドライバーを決定する．複数の同時代入はドライブ値から決定関数によって同一信号に対してだけなされるように決定され，1δ時間進めて最初に戻るということを繰り返している．ここでVHDLを複雑にしているものとして，ドライブ値と実効値を区別して扱うか，ということがある．ある信号は1

つ以上のドライブ値を持ち得るが，実効値は1つだけである．従って，設計者はドライブ値から1つの実効値を決定する決定関数を記述しなくてはならない．本稿では，この決定関数を大域決定プロセス (*global resolution process*) という1つのプロセスとして扱う．

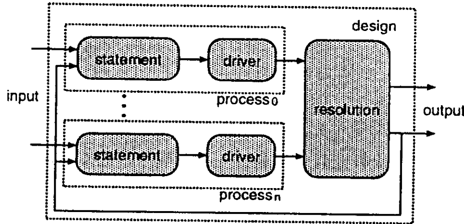

図 3: VHDL のプロセスモデル

3.4 VHDL における依存関係

VHDL には信号代入と変数代入がある．信号はハードウェア的な意味を持ち，信号に関連した時間的要素を持っている．これに対し，変数は主に動作記述中での中間値用に使用される．従って，変数代入は従来のプログラムスライシングにおけるデータ依存が適用できる．これに対し，信号代入は平行プロセスの間の唯一の通信手段であり，また，ソフトウェア言語にない遅延を伴うことから，新たな依存関係を導入する必要が出てくる．本稿では，従来のプログラムスライシングで提案されている制御依存やデータ依存に加えて，新たに信号依存 (*signal dependence*) という依存関係を導入する．VHDL におけるこれらの依存関係を以下のように定義する．

- 制御依存

  s_1を VHDL のプロセス p の文とし，s_2を同じ VHDL プロセス p 中の文とする．文 s_1から文 s_2へ**制御依存**があるとは，s_1は VHDL のフロー制御文であり，かつ，s_2が実行されるかどうかが s_1の実行結果に直接依存する場合をいう．

- データ依存

  s_1を VHDL のプロセス p の文とし，s_1は変数 vに対する代入とする．s_2を同じ VHDL プロセス p 中の文とし，s_2で変数 vが使用されるとする．文 s_1から文 s_2に**データ依存**があるとは，s_1で定義された変数 vが s_2に到達 (2.1節参照) する場合をいう．

- 信号依存

  s_1を VHDL のプロセス p の文とし，s_1は信号 Sigへの信号代入であるとする．また，s_2をあるプロセスの文とする．文 s_1から文 s_2へ**信号依存**があるとは，s_1の実行によってスケジューリングされるトランザクションにおいて，p が活性化され，かつ，s_2が実行されるようなものが存在する場合をいう．

3.5 スライシングの定義

スライシングにはスタティックスライシングやダイナミックスライシングだけでなく，さらにそれぞれに対して前向きスライシング (*forward slicing*)，後ろ向きスライシング (*backward slicing*) というものがあるが，本稿ではスタティックスライシングに限定してスライシングの概念の定義をする．

定義 3.1 VHDL 記述のスタティックスライスの判定基準とは2項のタップル $(s, Sig/V)$ である．この s は記述中の文で，Sig/Vは s で使用される信号，あるいは変数の集合である．与えられているスタティックスライスの判定基準 $(s, Sig/V)$ 上の VHDL 記述のスタティックスライス $SS(s, Sig/V)$ とは，s の実行の開始や終了，および s 中の実行結果に影響を与える記述中の全ての文からなる．□

与えられたスタティックスライスの判定基準上で VHDL 記述をスタティックスライスをするとは，その判定基準に関する記述のスタティックスライスを見つけることである．

3.6 スライシング例

例として取り上げる VHDL 記述を図 4に示す．この記述の 11 行目の 'x <= a' に関するスライスを求めてみる．スライシングの結果を図 5に示す．11 行目の 'x <= a' に依存関係を持たない記述部分 (5,8,12) が削除されているのがわかる．

これらの VHDL 記述の論理回路を図 6と図 7 に示す．この2つの図に明らかなように，図 6では信号 a と b のそれぞれにラッチがあるが，スライシングの結果，図 7のように信号 b のラッチが削除されて，ラッチが1つになっている．

```
    ENTITY example
      port (c,rst,clk : IN BIT;
            x,y       : OUT BIT);
    END example;
    ARCHITECTURE behavioral OF example IS
    SIGNAL a,b : BIT;
    BEGIN
 1    example: PROCESS(rst,clk,a,b,c)
 2    BEGIN
 3      IF (rst='0') THEN
 4        a <= '0';
 5        b <= '0';
 6      ELSIF (clk='1' AND clk'EVENT) THEN
 7        a <= a XOR c;
 8        b <= b XOR a;
 9      END IF;
10    END PROCESS example;
11    x <= a;
12    y <= b;
    END behavioral;
```

図 4: スライシング前

```
   ENTITY example
     port (c,rst,clk : IN BIT;
           x         : OUT BIT);
   END example;
   ARCHITECTURE behavioral OF example IS
   SIGNAL a,b : BIT;
   BEGIN
 1   example: PROCESS(rst,clk,a,c)
 2   BEGIN
 3     IF (rst='0') THEN
 4       a <= '0';
 6     ELSIF (clk='1' AND clk'EVENT) THEN
 7       a <= a XOR c;
 9     END IF;
10   END PROCESS example;
11   x <= a;
   END behavioral;
```

図 5: スライシング後

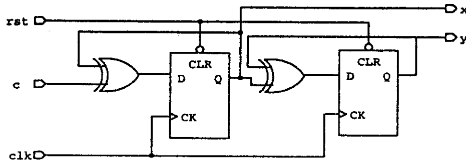

図 6: 図4のスライシング前の論理回路

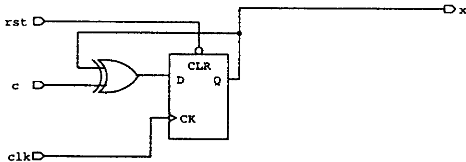

図 7: 図5のスライシング後の論理回路

3.7 VHDL スライシングの応用

プログラムスライシングはソフトウェア分野ではプログラムのデバッグ，テスト，保守，プログラムの合成など広範囲の応用例が示されている．これらと同様の応用がVHDL にも可能であると考えられる．これらの応用について簡単に述べる．

3.7.1 デバッグ

シミュレーションでエラーが見つかった時，その信号に関するスライスを求めることでエラーの原因となるドメインを狭めることができ，バグの位置を限定することができる．

3.7.2 テストパターンの生成

ある初期入力に基づいてシミュレーションを実行し，あらかじめ選択されているスライスを実行しないような分岐箇所に達した時点で，スライスにそって実行するように入力変数の値を変更していくという手続きを繰り返すことによって，機能検証用のテストパターンを生成することができる．これはソフトウェア分野ではダイナミックスライシングの技術によって実現される．

3.7.3 保守

システム中の1つのユニットを改造したり，削除したりする際に，変更が他のユニットに与える影響をスライシングで計算することができ，これにより他のユニットに影響を与えない変更を判定できる．

3.7.4 形式的検証

スライシングを利用して検証対象の箇所を限定して抽出し，検証問題のサイズを縮小するプロセスを自動化するなどの利用が考えられる．

4 VHDL 記述のグラフ表現

これまでのスライシング手法には，プログラムをグラフで表現しそのグラフに基づいてスライスを求めるものが多い．この手法の利点は，一度グラフ表現にしてしまえば，スライスを求めるという問題はグラフ表現上の点の到達可能問題とすることができ，比較的簡単なアルゴリズムでスライスを求めることができるということである．本稿においても，文献 [5] で提案されている制御フローグラフ (CFG:Control Flow Graph) と definition-use グラフ (DUG:Definition-Use Graph)，プロセス依存グラフ (PDG:Process Dependence Graph) をもとにして，新たに VHDL 記述の表現として信号依存を導入したグラフ表現を定義する．文献 [5] は並列プログラムをグラフによって表現して，スライスを求める手法についての研究で，その中で，従来の制御依存やデータ依存に加えて，選択依存 (*selection dependence*)，同期依存 (*syncronization deoendence*)，通信依存 (*communication dependence*) という 3 つの依存関係が導入されている．これらの依存関係は，VHDL には非決定性がないために選択依存はなく，同期依存に関しては時間的要素を加味して信号依存を導入している．通信依存は同期依存に関連しているため導入していない．

グラフ表現にしてスライシングを行なう記述例を図8に示す．この記述は wired-or に対応した決定関数を持つデータパスの記述である．

4.1 諸定義

定義4.1 arc-classified 有向グラフとは n 個のタップル $(V, A_1, A_2, \ldots, A_{n-1})$ であり，全ての $(V, A_i)(i = 1, \ldots, n-1)$ が有向グラフで，かつ，$A_i \cap A_j = \Phi(i = 1, 2, \ldots, n-1, j = 1, 2, \ldots, n-1)$ である．**単純 arc-classified** 有向グラフとは，arc-classified 有向グラフで任意の $v \in V$ に対して $(v, v) \notin A_i(i = 1, \ldots, n-1)$ が成り立つ．□

```
    ARCHITECTURE datepath OF bus1 IS
    SIGNAL bs: wired_qit_vector(3 DOWNTO 0) BUS;
    SIGNAL x1,x2,z: qit_vector(3 DOWNTO 0);
    BEGIN
1     b1: BLOCK(sel='00' OR sel='01')
2         BEGIN bs <= GUARDED a;END BLOCK;
3     b2: BLOCK(sel='10')
4         BEGIN bs <= GUARDED b;END BLOCK;
5     t <= bs
6     x1 <= NOT t;
7     x2 <= t+1
8     WITH sel SELECT
9       z <= x1 WHEN '00',
10           x2 WHEN '10',
11           '0000' WHEN OTHERS;
    END datapath;
```

図 8: データパスの VHDL 記述

定義 4.2　重み付き arc-classified 有向グラフとは $n+1$ 個のタップル $(V, A_1, A_2, \ldots, A_{n-1}, g)$ である．$(V, A_1, A_2, \ldots, A_{n-1})$ は arc-classified 有向グラフであり，$g : A_i \rightarrow L$ は弧に付けられる重みを指定する関数である．□

定義 4.3　有向グラフ (V, A) または arc-classified 有向グラフ $(V, A_1, A_2, \ldots, A_{n-1})$，重み付き arc-classified 有向グラフ $(V, A_1, A_2, \ldots, A_{n-1}, g)$ の**パス** (*path*) とは，弧の連続 $(a_1, a_2, \ldots, a_\ell)$ であり，a_iの終点が $a_{i+1}(1 \leq i \leq \ell - 1)$ の始点である．ここでは $a_i \in A(i \leq i \leq \ell)$ または $a_i \in A_1 \cup A_2 \cup \ldots, \cup A_{n-1}(1 \leq i \leq \ell)$ である．そして，$\ell(\geq 1)$ をパスの長さ (*length*) という．a_1 の始点が v_Iで a_ℓの終点が v_Tならば，そのパスは v_Iから v_Tまでのパス，省略して $v_I - v_T$と呼ぶ．□

定義 4.4　u と vを任意の 2 点とする．vから t への全てのパスが u を含む時かつその時に限り，u は vを**前方支配する** (*forward dominates*) という．u は vを前方支配し，$u \neq v$の時かつその時に限り，u は vを**真性前方支配する** (*properly forward dominates*) という．u は vを前方支配し，u を含む vから t への全てのパスが k以上の長さであるような整数 $k(k \geq 1)$ が存在する時かつその時に限り，u は vを**強前方支配する** (*strongly forward dominates*) という．u が vから t への全てのパスの中で vを真性前方支配する最初の点である時かつその時に限り，u は vの**直接前方支配者** (*immediate forward dominator*) と呼ぶ．□

4.2　制御フローグラフと definition-use グラフ

制御フローグラフ (CFG) と definition-use グラフ (DUG) は VHDL 記述の文を点，制御の流れを弧 (*arc*) によって表した arc-classified 有向グラフである．

定義 4.5　制御フローグラフ (*control flow graph*:CFG) は 11 個のタップル $(V, P_F, P_J, P_R, A_C, A_{P_F}, A_{P_J}, s, t, s_P, t_P)$ であり，$(V, A_C, A_{P_F}, A_{P_J})$ は単純 arc-classified 有向グラフである．$P_F \subset V$は**並列実行分岐点**，$P_J \subset V(P_F \cap P_J = \Phi)$ は**並列実行接合点**と呼ばれる．$P_R \subset V$は**決定関数点**と呼ばれ，任意の $v \in P_R$が $(\text{in-degree}(v) \geq 1) \wedge (\text{out-degree}(v) = 1)$ であることが成り立つ．$s \in V$は**開始点**と呼ばれる唯一の点で，入次数 in-degree(s)=0 が成り立つ．$t \in V$は**終端点**と呼ばれる唯一の点で，出次数 out-degree(t)=0 かつ，$t \neq s$ が成り立ち，任意の $v \in V$に対して s から vと vから t にそれぞれ少なくとも 1 つのパスが存在する．任意の弧 $(v_1, v_2) \in A_C$は**制御弧**，任意の弧 $(v_1, v_2) \in A_{P_F} \cup A_{P_J}$は**並列実行弧**と呼ばれる．$s_P \in V$ は**プロセス開始点**と呼ばれ，入次数 in-degree(s_P)=1 かつ任意の $v \in V$に対して $(v, s_P) = A_{P_F}$であることが成り立つ．$t_P \in V$は**プロセス終端点**と呼ばれ，出次数 out-degree(t_P)=1 かつ任意の $u \in V$に対して $(t_P, u) = A_{P_J}$であることが成り立ち，かつ $s_P \neq t_P$である．□

定義 4.6　definition-use グラフ (*definition-use graph*: DUG) は 9 個のタップル $(G_C, \Sigma_V, D, U, \Sigma_{Sig}, \Sigma_{Res}, \Sigma_F, S, R)$ である．$G_C = (V, P_F, P_J, P_R, A_C, A_{P_F}, A_{P_J}, s, t, s_P, t_P)$ は CFG であり，Σ_Vは変数と呼ばれる記号の有限集合，Σ_{Sig}は信号と呼ばれる記号の有限信号，$\Sigma_{Res} \subseteq \Sigma_{Sig}$は被決定信号と呼ばれる記号の有限集合，そして，$\Sigma_F$は被決定関数の名前の有限集合である．$D : V \rightarrow P(\Sigma_V)$ と $U : V \rightarrow P(\Sigma_V)$ は V からΣ_Vのベキ集合への 2 つの部分関数である．$S : V \rightarrow P(\Sigma_{Sig})$ と $R : V \rightarrow P(\Sigma_{Sig})$ は VからΣ_{Sig}へのベキ集合への 2 つの部分関数である．ただし，$Sig \in \Sigma_{Sig}$が $Sig \in \Sigma_{Res}$となる時，$S : V \rightarrow \Sigma_F$，$R : P_R \rightarrow \Sigma_F$，$S : P_R \rightarrow \Sigma res$ となる．□

DUG は定義に影響している情報と信号の情報がある CFG と見なすことができる．決定関数は，その関数を持った被決定信号 (*resolbed signal*) に代入を行う時にその文に対応した点から決定関数を表す点に制御が移る 1 つのプロセスとして扱っている．

CFG の点に D，U，S，R を付け DUG を作成する時，具体的には次の操作を行う．

- 代入文の左辺に変数が現れた時は D にその変数名を，信号が現れた時は Rにその信号名を入れる．
- 変数の値が使用された時 U にその変数名を，信号の状態が参照された時 R にその信号名を入れる．
- 被決定信号に対する信号代入の場合，その関数名を Sに入れ，その関数に対応する決定関数点の R に同じ名前を入れ，この点の S に被決定信号の信号名を入れる．

図 8の記述の DUG を図 9に示す．

4.3　プロセス依存グラフ

プロセス依存グラフ (PDG) は VHDL 記述の文を点，文の依存関係を弧で表した重み付き arc-classified 有向グラフである．

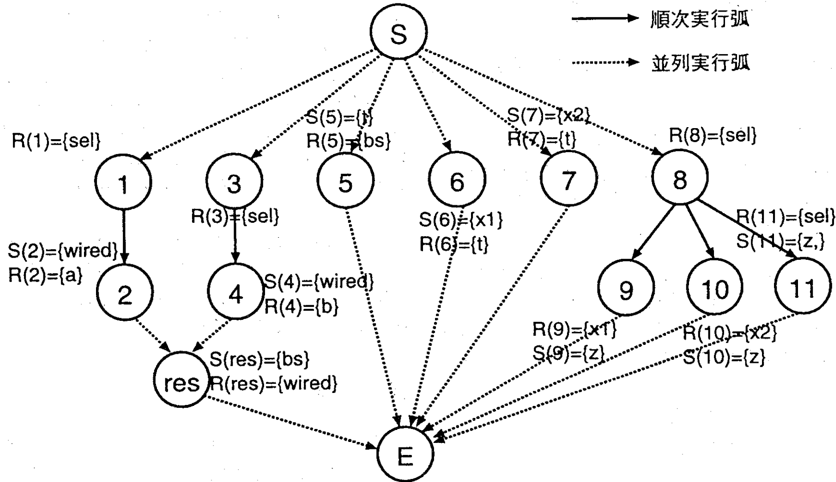

図 9: 図 8の DUG

定義 4.7 $(V, P_F, P_J, A_C, A_{P_F}, A_{P_J}, s, t, t_P, t_P)$ を CFG とし，$u \in V, v \in (V - (P_F \cup P_J))$ をグラフ上の任意の 2 点とする．vから u へのパス $P = (v_1 = v, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n = u)$ が存在し，P が vの直接前方支配者を含む時，Pの中に点 v'が存在しないで v'から u へのパスが v'の直接前方支配者を含まない時かつその時に限り，u は vに対する**直接強制御依存** (*directly strongly control-dependence*) である．2 つの後方部分 v' と v''を vが持ち，v から u へのパス $P = (v_1 = v, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n = u)$ が存在し，Pの中の任意の点 $v_i (1 < i \leq n)$ か v'を強前方支配し，v'' を強前方支配しない時かつその時に限り，u は vに対する**直接弱制御依存** (*directly weakly control-dependence*) である．□

定義 4.8 $(G_C, \Sigma_V, D, U, \Sigma_{Sig}, \Sigma_{Res}, \Sigma_F, S, R)$ を VHDL 記述の DUG とし，u と vをグラフ上の任意の 2 点とする．$(D(u) \cap U(u)) - D(P') \neq \Phi (D(P') = D(v_2) \cup \ldots \cup D(v_{n-1}))$ であるような u から vへのパス $P = (v_1 = v, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n = u)$ が存在する時かつその時に限り，u が vに対する**直接データ依存**である．□

定義 4.9 $(G_C, \Sigma_V, D, U, \Sigma_{Sig}, \Sigma_{Res}, \Sigma_F, S, R)$ を VHDL 記述の DUG とする．G_Cは記述の CFN$(V, P_F, P_J, P_R, A_C, A_{P_F}, A_{P_J}, s, t, s_P, t_P)$ であり，u と v をグラフ上の任意の 2 点とする．$S(v) \cap R(u) \neq \Phi$が成り立つ時かつその時に限り，u は vに対する**直接信号依存**という．□

定義 4.10 プロセス依存グラフ (*process dependence graph*: PDG) は重み付き arc-classified 有向グラフ (V, Con, Dat, Sig, g) である．ここでの Vは VHDL 記述の CFG の点集合である．Con は制御依存弧の集合であり，u が vに対する直接弱制御依存である時かつその時に限り，任意の $(u, v) \in Con$ が成り立つ．Dat はデータ依存弧の集合であり，u が vに対する直接データ依存である時かつその時に限り，任意の $(u, v) \in Dat$ が成り立つ．Sigは信号依存弧の集合であり，u が vに対する直接信号依存である時かつその時に限り，任意の $(u, v) \in Sig$が成り立つ．また，$g : Sig \rightarrow L$ は信号依存弧に付けられる重みを指定する関数である．□

PDG の信号依存弧には重みが付けられている．この重みとは信号代入の際の遅延の情報を表したもので，本稿では実時間遅延を扱っていないため，この信号依存弧には 1δ の重みが付けられることになる．

図 9から導出した PDG にを図 10に示す．

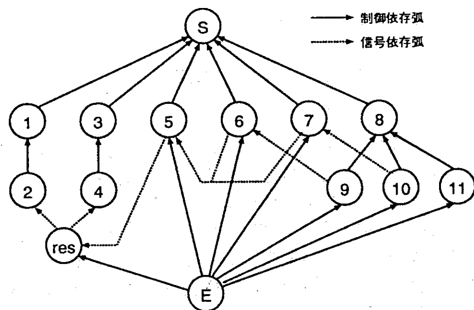

図 10: 図 8の PDG

5 スライシングアルゴリズム

(スタティック) スライシングにはPDGを使用する．以下にそのアルゴリズムを示す．

- 注目する点を始点とするパスを全て残す
- パスの中にプロセスやループの始点が含まれる場合，その始点に対応するループやプロセス等の順次本体部分の終点を残す

このアルゴリズムに従って，実際に図10を使ってスライシングを行なってみる．今回は例えばスタティックスライスの判断基準 (9, z) に関するスタティックスライスを求めると，図11のようなグラフが得られる．しかし，このグラフは必ずしも最小のスライスであるとはいえない，というのは (9, z) の信号代入が行なわれるには，‘sel=’00’’ でなければならないため，2で設定された値がzに到達する間に変化しなければ，実際には3と4も削除できるはずである．同様に次のような場合を考える．

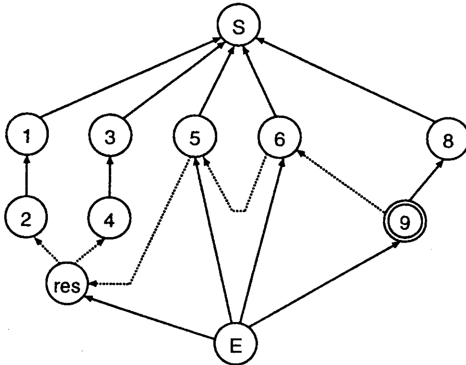

図 11: (9, z) に関するスタティックスライス

```
s1 :    a <= 'run';

s2 :    IF (a = 'reset') THEN
s3 :        b <= c;
```

s_1と s_2は異なるプロセス文中にあると仮定する．この場合，s_1から s_2には信号依存がある．しかし，s_1によって s_3のトランザクションがスケジューリングされることはないので，s_1が s_3の実行に直接影響を与えない．さらに信号属性についても同様の議論ができる．このような信号の状態や属性を考慮した場合，より精度の良いスライスを求めることができると考えられる．

6 おわりに

本稿では，デルタ遅延モデルに基づいて VHDL の仕様に限定を加えた上で，グラフ表現によって VHDL 記述を表し，そのグラフ上でのスライシングアルゴリズムを示した．VHDL では信号代入によって平行プロセス間の通信および遅延を表現するため，新たに信号依存という依存関係を導入し，グラフ上では信号依存弧という重みが付けられた弧によって表現した．こうすることでにより，スライシングの際にグラフを探索する時，探索するパスに制約を加えることにより，スライスをより限定することも可能である．

今後の課題としては，今回加えた制限への拡張を行っていくとともに，信号の状態による参照可能性を解析することによって，より精度の良いスライスを求めるアルゴリズムを定めていくことである．

参考文献

[1] 野村雅也, 岩井原瑞穂,“プログラムスライシングを用いた HDL 記述の検証,” 情報処理学会研究報告 DA-75-2,1995 年.

[2] C.D.Kloos, P.T.Breuer(ed.), “*Formal Semmantics for VHDL*,” Kluwer Academic Publishers, 1995.

[3] M.Weiser, “Program Slicing,” *IEEE-CS Transactions on Software Engineering*, Vol.SE-10, No.4, pp.352-357, 1984.

[4] 下村隆夫, “Program Slicing 技術と応用,” 共立出版株式 会社, 1995 年 7 月.

[5] J.Cheng, “Slicing Concurrent Programs - A Graph-Theoretical Approach,” *Proc. 1st Int. Workshop on Automated and Algorithm Debugging (AADEBUG)*, pp.223-240, 1993.

[6] Z.Navabi, “*VHDL:Analysis and Modeling of Digital System*,”
McGraw-Hill, Inc., 1993; (訳)“VHDL の基礎,” 日経 BP 出版センター, 1994 年 8 月.