設 計 自 動 化 75－2
（1995．5．26）

プログラムスライシングを用いたHDL 記述の検証

野村 雅也　岩井原 瑞穂

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: {mnomura, iwaihara}@is.kyushu-u.ac.jp

本論文では，ソフトウェア工学の手法であるプログラムスライシングをハードウェア記述に適用させることを提案する．プログラムスライシングはプログラム中の特定の動作について，依存を持つすべての記述を抽出することにより動作の解析，変更を支援する技術である．ハードウェア記述のプログラムスライシングは機能検証やテスト生成，設計者支援，記述の合成など広範囲の応用が考えられる．本稿ではハードウェア記述言語 VHDL にプログラムスライシングを適用した例を示し，いくつかの問題点を述べる．

和文キーワード　プログラムスライシング，依存解析，ハードウェア記述言語，VHDL，設計検証

Program Slicing on Hardware Descriptions and its Application to Design Verification

Masaya NOMURA　Mizuho IWAIHARA

Department of Information Systems
Interdisciplinary Graduate School of Engineering Sciences
Kyushu University
6-1 Kasuga-koen, Kasuga-shi, Fukuoka 816 Japan

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

We propose applying program slicing to hardware descriptions; program slicing is a technique developed in the software engineering field, and it can extract all sentences which have a certain dependence to a given sentence, and it can be utilized for analysis and modification of programs. Program slicing for hardware description languages should have a variety of applications in VLSI CAD, such as functional verification, test generation, designer assist, and design synthesis. We show several examples of program slicing on VHDL descriptions, and address a list of related issues.

keywords program slicing, dependence analysis, hardware description language, VHDL, design verification

1 はじめに

ハードウェア記述言語 (HDL) による自動論理合成の普及は論理回路開発期間の大幅な短縮を実現しており, HDL は抽象度の高い記述から論理回路までの一貫したシステム記述言語としての役割を期待されている.

今後, ハードウェア記述言語による設計資産の蓄積・巨大化が予想されるなかで, ハードウェア記述の設計検証, 保守, 再利用などの問題は重要さを増すと思われる. ソフトウェア工学の分野では, これらの問題に対し, プログラムスライシングと呼ばれる手法に基づく研究が多くなされている. プログラムスライシングはプログラム中の文の間の依存関係を解析する手法であり, 1982 年に Weiser によって提案された [9][11].

本稿では, ハードウェア記述言語にプログラムスライシングを適用することを提案し, その応用例としてデバッグ, テスト生成, 設計検証への適用を議論する. 設計検証においては, ハードウェア記述から検証に必要な箇所を抽出し, 検証対象の回路規模を縮小することへの応用が挙げられる. さらに VHDL を対象としたプログラムスライシングの例を挙げ, スライス計算におけるハードウェア記述言語特有の問題として, 遅延や不完全指定, 並列性などを議論する.

本稿ではまず 2節でプログラムスライシングについて述べ, 3節で VHDL に対するスライシングの定義および例を挙げる. 次に 4節で VHDL のスライシングの応用および関連研究について述べ, 5節で VHDL におけるスライス計算の問題点をまとめる. 6節はさらに VHDL とソフトウェアのスライシングの異なる点を述べ, 最後に 7節でまとめを述べる.

2 プログラムスライシング

プログラムスライシング [9][11] (以降, 単にスライシング) は, 大規模なプログラムの動作を解析, 修正する際, 注目すべき動作の記述を取り出すことによりプログラムを簡約するソフトウェア工学の手法である. スライシングにより求まるプログラム片は注目する動作のふるまいが保存されているため. プログラムの解析, デバッグ等に用いられ, その効果は広く認められている.

スライシングはプログラム内のある文の実行に影響を与えるすべての文を抽出する技術であり, 抽出された文の集合をスライスと呼ぶ. スライスには, プログラムを静的に解析して得られる静的スライス [11] と, ある入力に基づいてプログラムを実行したとき, ある文の実行に実際に影響を与えた, 文の集合である動的スライス [4] の二つがある. 静的スライスはある文の実行に影響を与える可能性のある文の集合であり, 以下本稿では静的スライスについて主に議論する. 逐次実行型のプログラムにおける静的スライスは以下のように定義される.

- **データ依存**
  文 s1 から文 s2 へのデータ依存関係があるとは, 文 1 における, ある変数 v の定義が, v を使用している文 s2 に到達する場合, すなわち,
  1. 文 s1 において変数 v を定義している (文 s1 で変数 v に値を設定している), かつ,
  2. 文 s2 において v を使用している (文 s2 で変数 v の値を参照している), かつ,
  3. v を再定義しない, 文 s1 から文 s2 への実行可能なパスが存在する場合である.

- **制御依存**
  文 s1 から文 s2 への制御依存関係があるとは, 文 s1 は if 文か while ループ文であり, 文 s2 の実行の有無が文 s1 の実行結果に直接依存する場合である.

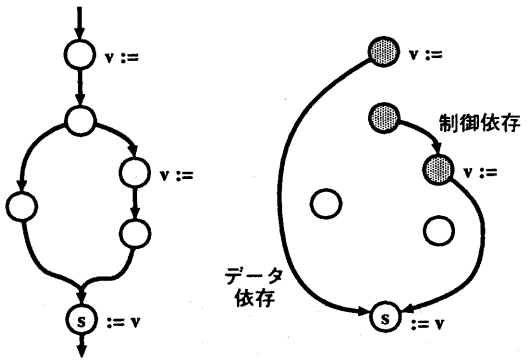

図 1: 文 s における変数 v に関する静的スライシングの求め方

文 s における変数 v に関する静的スライスとは, データ依存関係, 制御依存関係をたどって, 文 s の変数 v に到達するすべての文の集合である (図 1). 文の集合 $S = \bigcup_i S_i$ に関する静的スライスは, $\bigcup_i$ (S_i の静的スライス) で定義される. 静的スライスの例を図 2 に示す. 文 s の静的スライスは, 文 s の実行

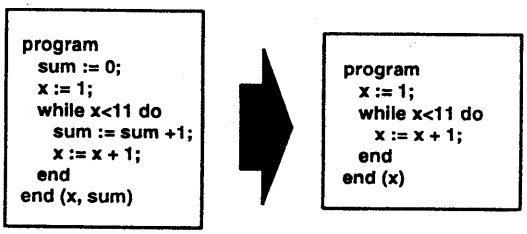

図 2: end 文における変数 x に関する静的スライシングの例 ([8] より)

結果を保存する完全なプログラムを構成する (もちろん, 必要なデータ宣言などを含める) という特徴を持つ.

スライスの品質として, 着目している文に影響を与える可能性がある文だけが求められるのが理想であるが, 解析の複雑さから実際には影響を与えない文もスライスに入りうる. そのためできるだけ文の数が少ないスライスほど精度が高いといえる.

3 VHDL におけるスライシング

3.1 スライシングの定義

代表的なハードウェア記述言語である VHDL において, その記述のスライシングを直感的に定義する. VHDL では, 一般のプログラミング言語のようにプログラムが制御文に従い 1 文づつ実行されるという意味ではないため, 「ある代入が実行される」や, 「ある制御文により実行制御が変った」などの概念はそのままあてはまらない. VHDL の代入文はイベント駆動であり, 右辺の信号が変化したとき右辺が評価され, 左辺の信号に指定時間後に代入が行なわれる. そのため VHDL のある代入文 $s1$ に対する静的スライシングとは,

- ある代入文 $s1$ の右辺の信号または変数を変化させるイベントの発生に影響を与える文の集合

と定義できる. 同様に VHDL における動的スライシングを定義すると,

- ある与えられた入力系列のもとで, 代入文 $s1$ の右辺の信号または変数をある時刻に変化させたイベントの発生に影響を与えた文の集合

となる.

制御依存に対しては, VHDL における制御文として, LOOP 文および PROCESS 文本体における順次実行の繰り返し等があげられる.

3.2 適用例

具体的な VHDL の記述に対して, スライシングを適用してみる.

図 3に例題を挙げる. この回路は 2 つの D フリップフロップの出力を 2 入力セレクタが sel の値により選択, out に出力する回路である. (a) は D フリップフロップの動作記述, (b) は 2 入力セレクタの動作記述, (c) は例題回路の構造的記述である. これらの VHDL 記述よりのセレクタの信号線 a の動作に注目するため, $out <= a$ の信号線 a についてのスライスを求める.

スライスを求める手順は以下のようにする.

1. プロセス $p1$ 内についてのスライスを求める.
2. 1 の結果, 構造的記述より $p1$ と依存関係を持つと考えられるプロセス $p2$ についてのスライシングを求める.

1, 2 の手順をプロセス間の依存関係がなくなるまでくり返す.

まず, スライスを考える文のあるプロセス *2_select* についてのスライスを求めると図 4.(b) のようになる. センシティビティリスト内の信号 b は使われない信号となるためリストより削除される.

2_select の記述は全入力に対する出力が明確に記述されており組合せ回路の記述となるが, スライスの結果 $sel =' 1'$ の場合の記述が削除される. この場合 $sel =' 1'$のときの動作が指定されていないため (不完全指定), スライス後の回路の論理合成を行なうと, x の値を保持するためのフィードバックが生じることに注意する.

2_select のスライシングによって依存を持つと判断される信号 (sel, a) について変化を与えるプロセスについてのスライシングを求める. 信号 sel については外部入力線となるため, これ以上スライスは求めない. (c) の構造的記述より信号 a の値は $d\_ff : u1$ の出力 q に接続されるため $d\_ff : u1$ について信号 Q についてのスライスを求める必要がある. $d\_ff : u2$ については *2_select* において信号 b が削除されたため, スライスする必要がなく削除される.

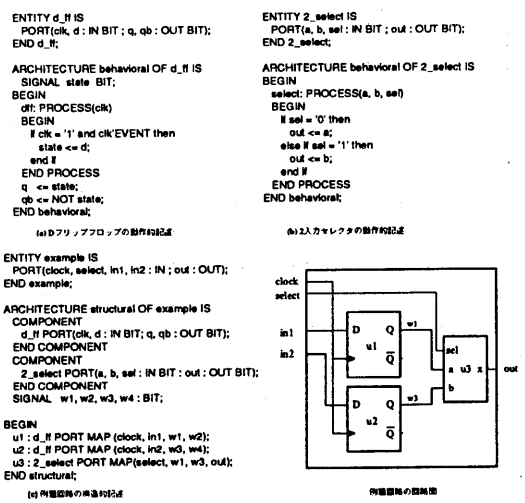

図 3: 例題の VHDL 記述と回路図

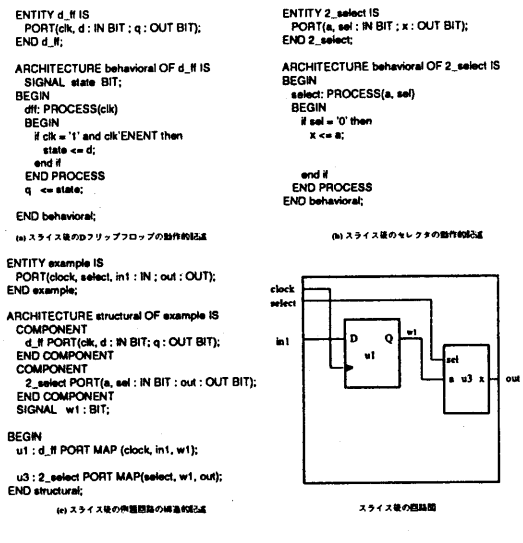

図 4: スライス後の例題の VHDL 記述と回路図

D フリップフロップの動作記述は 3 つのプロセス[1]から構成される．このうち，qの値の決定に直接依存するのは q <= $state$; である．q <= $state$; が実行されるためには信号 $state$ の値の変化を記述する文が実行されなくてはならないので，さらに信号 $state$ について他の 2 つのプロセスを調べる．プロセス d_ff には $state$ に変化を与える記述，$state$ <= d があり，この文について信号 d でスライスされる．

[1]本文においては，PROCESS 構文で囲まれない記述についても非明示的なプロセスと考える．

qb <= $state$ は $q, state$ の値の決定に依存は持たないためスライスされない．

以上，スライスにより削除された D フリップフロップの qb, セレクタの b, 構造的記述の $in2, w2, w3, w4$ の各信号，$d\_ff : u2$ を構造的記述より削除したスライスを生成する．スライシングにより生成された記述を図 4 に示す．スライス後の記述はフリップフロップの数が 2 つから 1 つへと減っていることがわかる．

4 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プログラムの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ハードウェア記述にも可能である．またハードウェア特有の問題に対しての応用が考えられる．以下では，これらの応用について議論し，また関連する研究についても述べる．

4.1 デバッグ

ある信号の値が誤っていることがわかった場合，その信号に値を設定した文に関するスライスを求めることにより誤った値を生成した箇所を限定することができる．

4.2 設計階層間の比較

VHDL のトップダウン設計においては，概念的な記述から動作レベル，合成レベルなどの設計階層を経る．設計階層を下る過程で混入する誤りを発見するために，ある特定の文についてのスライスを各階層で求め，設計者が比較することにより，文の依存関係が変わっていないかを確認できる．

4.3 テストパターンの生成

ある初期入力に基づいてハードウェア記述のシミュレーションを実行し，あらかじめ選択されていたスライスを実行しないような分岐箇所に達した時点で，スライスに沿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．その後，入力変数の値を参考に機能検証用のテストパターン (テストベクトル) が生成される．これにより，指定した文にイベントを発生させるテストパターンを生成することができる．

4.4 保守

例えば, 1つの設計がA, B, Cの三つの機能ユニットを持つとき, B, Cの機能は変えないで, Aの機能をA'に改造したい場合がある. このとき, B, Cの機能に影響を波及させないようにAを改造する. したがって, 改造後のテストは改造対象A'についてのみ行えば良く, 改造対象でない機能B, Cの再テストが不要になるという利点がある. 同様に, 1つの設計からある機能ユニットAを削除する場合, Aを起動する部分およびAから影響を受ける部分をスライスで求めることにより, 変更すべき箇所が求まる.

4.5 情報抽出

ハードウェア記述から種々の有用な情報を抽出する手法として, 命令セットプロセッサのコンパイラを生成するための情報抽出[1]や, テスト生成のための情報抽出がある[10]. しかしこれらの手法はそれぞれの目的に特化しており, 一般化するには考慮すべき点が残されている. プログラムスライシングはコンパイラのための様々な情報抽出に用いられており, ハードウェア記述においても一般的な枠組みを与えることが期待できる.

4.6 形式的検証

形式的検証は近年2分決定グラフを用いたCTLモデルチェッキングなどの手法の発展により[5], 実用化が大きく進んでいる. しかし現在においても検証可能な回路規模はまだ小さく, 大規模な回路を検証する場合, 抽象化などの手段を用いて, サイズを縮小化する必要がある[6]. このプロセスはまだ人手に頼る部分が多く, この過程で誤りが混入する恐れがある.

さらにハードウェア記述を検証用の別の言語に翻訳する手間も問題である. VHDL自体を検証用言語として用いれば, このような問題が生じないのであるが, VHDLは公式的なセマンティクスが定まっていないため, 形式的検証のためのセマンティクスの確立が試みられている[12][7].

望まれる形式的検証システムの要件として, VHDL記述にできるだけ人手を加えずに検証を行なうことが可能であり, しかも検証の対象となる箇所を限定することにより, ハードウェア記述をできるだけ縮小することが必要である. このように検証のためにハードウェア記述を縮小する, つまり検証の対象となる機能を抽出する手法として, プログラムスライシングを適用することが考えられる.

VHDL ベースの形式的検証システムとしてPrevail[2]があるが, そこでは検証の対象となるコンポーネントを指定し, その記述と仕様の記述をともに検証用言語に変換して等価性を比較しており, コンポーネント内部まで解析して無関係な記述を削除する操作は行なわれていない.

プログラムスライシングを活用した形式的検証手法として, 以下のモデルが考えられる. まず, 論理合成可能なハードウェア記述から, 検証を行なうべき動作に対応する文を検証者が指定し, その文からスライスを求める. 例えば, マイクロプロセッサの記述において, ADD命令の加算を行なっている文を指定し, ADD命令を実行するために必要な部分をスライスとして求める. その後, スライシングされた記述をさらに抽象化し, 検証可能なサイズの記述を生成し, ハードウェア記述が仕様を満たしているかを判定する. このような検証システムの例として, 図5にCTLモデルチェッキングを使用した場合の検証のフローを示す.

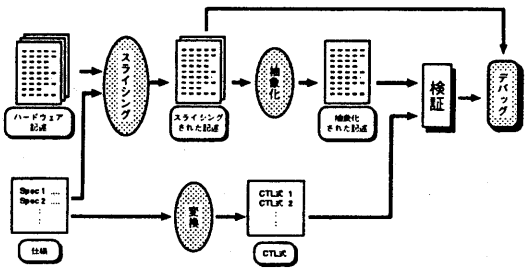

図 5: 検証のフロー

5 VHDLにおけるスライス計算

VHDLにおいてスライスを計算するときに考慮すべき基礎的な問題について, 例を用いて説明する.

5.1 順次文

まず図6のプロセスの内部の順次文について検討する.

VHDLの順次文はIF文やLOOP文などによってソフトウェアに似たプログラム・フロー制御が行われるが, 様々なタイミングの概念を導入しているた

め, ソフトウェアのスライス計算をそのままあてはめることはできない.

```
(1)  exmpl: PROCESS
(2)  BEGIN
(3)      WAIT UNTIL clk = '1';
(4)      IF sel = '1' THEN
(5)          out <= a;
(6)      ELSE out <= b;
(7)      END IF;
(8)  WAIT UNTIL recieved = '1';
(9)  out <= 0;
(10) END
```

図 6: プロセス内部のスライシングの例

5.2 保留

順次文においては, 各文は 0 時間で評価されるが, WAIT 文やプロセス文の最後にプログラム・フローが到達すると, 保留状態になる. VHDL では, 各イベントが発生する時刻も意味を持つため, WAIT 文により同期や遅延が生じると, WAIT 文に続く文にイベントが生じる時刻が影響を受けると考えられる. そのため WAIT 文に続く文のスライスには, その WAIT 文が含まれなければならない. 例えば, 図 6 の文 (9) のスライスは, {1, 2, 3, 8, 9, 10} となる.

5.3 不完全指定

図 6において, 文 (5) に対するスライスを求めると {1, 2, 3, 4, 5, 7, 8, 10} となり, 文 (6) はスライスから除外される. ところが, このスライスでは sel='0' のときの out の値が指定されない不完全指定となる. この場合, 通常の合成系では out の値を保持するラッチが合成される. もともと組合せ回路として記述したプロセスのスライスを合成すると, ラッチが生じるようになるのは回路の性質を大きく変えるという点で望ましくない. ソフトウェアの場合, 変数はもともと記憶素子の意味も持っているため, 上記のような問題は生じない.

5.4 信号の属性

VHDL ではクロックによる同期を明示的に表現されるが, クロックの立ち上がりエッジの表現として, 例えば以下のような信号の属性による条件式が用いられる.

```
IF ( clk ='1' AND NOT clk'STABLE)
```

このように信号の属性を用いることにより, 信号が時系列に沿って変化する波形を条件とできるのであるが, 通常のプログラミング言語ではこのような記述はできない. このような条件式をスライス計算の対象にすることもできる. 例えば上の条件式についてスライスを行う場合, この式に当てはまらないイベントを生じる式, 例えば clk の立ち下がりを発生する式は, スライスから除くことができる. また, 指定された文からそこに到達することができるクロックへの同期を表現する文までのパスを求めることにより, 1 クロック分の動作を抽出することができる.

5.5 制御依存およびモジュール間の依存

図 6において, 文 (5) に対するスライスには文 (3) と文 (8) が含まれていたのであるが, これはサブプログラムではないプロセスの文部においては, プログラム・フローが最後の順次文に達すると, 最初の文に戻るという無限ループになっていることによる. このようにプロセスの順次文ではソフトウェアと同様の制御依存を考慮する必要がある.

また文 (5) のスライスの計算において, スライスに文 (9) が含まれていなかったのは, 入力 a および sel と出力 out の間には依存関係が存在しないことを仮定していたためであった. プロセス exmpl が単独で動作するならばこの仮定は成り立つが, exmpl がある回路の一部であるならば, exmpl の外部においても a および sel と out の依存関係を調べる必要がある.

5.6 データパス

最後にデータパスをスライスする問題について取り上げる. 図 7の記述では, セレクタ信号 sel によって選ばれた信号が 4 値の 4 ビットバス bs に wired_or で書き込まれ, そのバスは (10) 行以下の記述で再び sel の値によって分離されている. この記述のようにハードウェアでは 1 つのバスがいくつかの機能で共用されることが行われるが, スライスの計算では各機能ごとに使われるデータパスを分離できることが必要である.

例えば, 図 7の文 (13) についてスライスを計算する場合, 信号 x1 とさらにそれが依存している信号 t が求められる. t に影響を与える bs までをさらに求めると, 結局スライスは {4, 5, 6, 7, 8, 10, 13} となる. ところが, (13) が実行される場合, sel の値は 00 に束縛されていることから, (7) も削除できることに

なり，スライスは {4, 5, 6, 7, 8, 10, 13} となる．ただしここで sel の値は，(5) で設定された値が (13) に到達するまでの遅延 20ns の間変化しないという仮定が必要なことに注意しなければならない．

```
(1)  ARCHITECTURE datapath OF bus1 IS
(2)  SIGNAL bs : wired_qit_vector (3 DOWNTO 0) BUS;
(3)  SIGNAL x1, x2, z : qit_vector (3 DOWNTO 0);
(4)  BEGIN
(5)      b1: BLOCK (sel ='00' OR sel = '01')
(6)          BEGIN bs <= GUARDED a; END BLOCK;
(7)      b2: BLOCK (sel ='10')
             BEGIN bs <= GUARDED b; END BLOCK;
(8)      t <= bs AFTER 10ns;
(9)      —
(10)     x1 <= NOT t AFTER 10ns;
(11)     x2 <= t + 1 AFTER 10ns;
(12)     WITH sel SELECT
(13)         z <= x1 WHEN '00',
(14)         x2 WHEN '10',
(15)         '0000' WHEN OTHERS;
(16) END datapath;
```

図 7: データパスのスライシングの例

6 ソフトウェアのスライシングとの相異点

前節までに述べてきた VHDL 記述のスライスを計算するうえでの，VHDL に特有の問題とソフトウェアと同様となる問題について，さらに議論を進める．

6.1 実時間性および遅延

多くのソフトウェア言語は実時間の遅延を記述することはできず，変数への代入は代入文が実行された直後に終了する．しかし VHDL においては，信号への代入は代入文が実行されると，信号代入のイベントがスケジューリングされ，実際の信号の値の変化はデルタ遅延あるいは実時間の遅延を伴う．さらにプロセス内の各代入文は同一のシミュレーション時刻内で実行されるので，代入文直後の文の実行時点において信号は新値を得ていないこととなる．

また EVENT などの信号属性も VHDL 特有の表現である．これらは 5.6節で述べたように，解析を複雑にすると考えられ，VHDL に形式的意味を与えるうえでも問題になっている [2][12]．解析を容易にするために，AFTER 節などによる遅延を無視しデルタ遅延のみを扱うことが考えられる．

6.2 並列実行

VHDL は同時文によりステートメントレベルでの並列実行が可能である．Ada などの並行プログラミング言語におけるプロセスの依存関係の解析も研究がなされている [3]．VHDL との大きな違いとしては，一般の並行プログラミング言語においては入力データに応じて 1 つのプロセス記述から任意の数の平行なプロセスが生じ得るが，VHDL においては多くの場合，プロセスはハードウェア資源に結び付けられており，平行に動作し得るプロセスの数が決まっていることである．このため VHDL の依存関係の解析はソフトウェアの場合より容易になるであろう．

ただし VHDL の場合でもトップダウン設計においては，概念的で具体的なハードウェア資源へのマッピングが行われていない場合などには，ソフトウェアと同様の複雑さを持つことも有り得る．

6.3 合成可能な記述

VHDL では，すべての正しい記述が必ずしも論理合成可能ではない．スライシングによって合成可能性が損なわれることがある．それは例えば完全指定であった条件代入文がスライスにより不完全指定になる場合である（3.2節の例）．スライスの結果を論理合成しゲートレベルで検証する場合は，合成可能性を保存する必要がある．この場合，不完全指定のケースに陥る入力は与えられないと仮定して，合成後の回路が小さくなるように不完全指定のケースに任意の動作を割り当てることが考えられる．

またデバッグなどで影響を与える箇所を求めるだけであるならば，合成可能性を考慮する必要はない．この場合は，削除される部分を assertion 文に置き換える方法などが考えられる．

6.4 動作記述と構造記述

VHDL の記述は大別して次の二つの記述構造を持つ．

- 動作的記述
  一般に ARCHITECTURE 文内で記述され，信号，変数への代入文を IF 文，CASE 文などが制御する形式で表される．
- 構造的記述
  ENTITY や COMPONENT といった動作的記述間の接続を静的に記述する．

このうち動作的記述内のスライシングにより必要な信号が判別できれば，不必要な信号に接続され，不必要な信号の値の決定に関連する構造的記述は削除することができる．そのため構造的記述におけるスライシングは比較的容易である．

7 まとめ

ハードウェア記述言語がシステム設計の中核となり，ハードウェア記述言語による設計資産が増大してゆくなかで，今後ソフトウェア工学的手法を用いたハードウェア記述の管理が重要になってゆくと思われる．本稿では，ソフトウェア工学において幅広い応用を持つ，プログラムの解析手段であるスライシングをハードウェア記述言語に応用することを提案し，様々な実際的問題に有効であることを述べた．そして代表的ハードウェア記述言語のひとつであるVHDLにプログラムスライシングを適用する場合の基本的問題点について整理を行なった．今後，VHDLに対するスライシング・アルゴリズムをより具体化し，システムの開発を行なってゆく予定である．

謝辞

日頃ご討論頂く九州大学 大学院総合理工学研究科 安浦寛人教授，村上和彰助教授ならびに安浦研究室の諸氏に感謝致します．

参考文献

[1] Akaboshi, H., H. Tomiyama and H. Yasuura, "Compiler Generation from Hardware Description Language, " *Proc. 1st Asian Pacific Conf. Hardware Description Languages, Standards & Applications*, pp. 76–78, Dec. 1993.

[2] Borrione, D. D., L. V. Pierre and A. M. Salem, "Formal Verification of VHDL Descriptions in the Prevail Environment, " *IEEE Design & Test of Computers*, pp. 42–55, June 1992.

[3] Cheng, J., "Slicing Concurrent Programs—a Graph-Theoretical Approach, " *1st Int. Workshop on Automated and Algorithmic Debugging (Lecture Notes in Comp. Sci.*, pp. 223–240, 1993.

[4] Kore, B. and J. Laski, "Dynamic Program Slicing, " *Information Processing Letters*, Vol. 29, No. 10, pp. 155–163, Oct. 1988.

[5] 平石 裕実, 浜口 清治, "論理関数処理に基づく形式的検証手法, " 情報処理 Vol. 35, No. 8, pp. 710–717, 1994 年 8 月.

[6] 藤田 昌宏, 陳 奔, 山崎 正実, "形式的検証手法の実設計への適用例, " 情報処理 Vol. 35, No. 8, pp. 719–725, 1994 年 8 月.

[7] Mermet, J. (ed.), "*VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, " Kluer Academic, 1992.

[8] Z. ナビバ, "VHDL の基礎, " 日経 BP 出版センター, 1994 年 10 月.

[9] 下村隆夫, "Program Slicing 技術とテスト，デバッグ，保守への応用, " 情報処理 Vol. 33, No. 9, pp. 1078–1086, 1992 年 9 月.

[10] Vishakantaiah, P, J. Abraham and M. Abadir, "Automatic Test Knowledge Extraction from VHDL (ATKET), " *Proc. 29th ACM/IEEE Design Automation Conf*, pp. 273–278, June 1992.

[11] Weiser, M., "Program Slicing, " *IEEE Trans. Software Eng.*, Vol. SE-10, No. 4, July 1984.

[12] Xin Hua, G. and H. Zhang, "Formal Semantics of VHDL for Verification of Circuit Designs, " *Int. Conf. Computer Design*, pp. 446–449, 1993.