論理検証のための
モジュール間インタフェース仕様記述とその利用法

岩下 洋哲*　古渡 聡†　長井 寛志†

*富士通研究所　†富士通

〒211-8588 川崎市中原区上小田中 4-1-1
hiroaki@flab.fujitsu.co.jp

ハードウェアモジュールの網羅的な論理検証のためには，モジュール間のインタフェースを数学的に規定する仕様記述が必要である．さらに，その仕様記述を様々な検証技術で共通して利用できる枠組みの実現が望まれている．本文では，モジュール間インタフェースに適した仕様記述方式を提案し，仕様記述からそれに対応した順序機械への変換によりポータビリティの高い検証モデルを実現するアプローチを，実設計への適用例を交えて紹介している．

論理検証，モジュール検証，インタフェース仕様，正規表現，有限オートマトン

Practical Use of Module Interface Specifications for Design Verification

Hiroaki Iwashita*　Satoshi Kowatari†　Hiroshi Nagai†

*Fujitsu Laboratories Ltd.　†Fujitsu Ltd.

4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki 211-8588
hiroaki@flab.fujitsu.co.jp

It is indispensable for comprehensive verification of hardware modules to provide formal interface descriptions of the modules. The interface descriptions should be shared by various verification techniques. We introduce a language suited for module interface description, and show a method to generate portable verification frameworks, which is a finite state machine written in HDL, from the interface descriptions. A case study in actual hardware verification is also presented.

verification, module verification, interface description, regular expression, finite automaton

1　はじめに

ハードウェア設計において，論理検証にかかる工数は全体の中でも大きな比率を占めるようになっている．対象回路規模の増大に対して論理検証のコストは指数的に増大する．しかしながら設計開始から製造までの間に費やすことのできる時間は限られており，結果として，検証の質が低下していくことが懸念されている．難しさが規模に対して指数的に増大するに問題に対しては分割統治の解法が必要であることは，多くの人が認めるところである．論理検証の分野では，モジュール検証を効率よく実施することが必要，ということになる．

検証の自動化を実現するためには，解くべき検証問題が明確に定義されている必要がある．その意味で，モジュール検証における問題は，正しい動作と入力制約を明確に定義することが難しい点にある．モジュールの正しい動作が保証されるのは，一般に特定の制約条件（入力制約）に違反しないパターンが入力された場合のみである．モジュールは周りの他のモジュールと協調して動作するものであるため，現実の設計では，単体としての正しい動作や入力制約を単純には規定できない場合が多い．

特にモジュール間のインタフェース仕様は，モジュールの動作や入力制約を規定する上での鍵となる．モジュールを外部から見た仕様はインタフェース仕様を使って定義される．入力制約は入力に関するインタフェース仕様と関連が深い．

そこで我々は，現実的なインタフェース仕様をより易しく記述する方式の開発を目指している．その際，仕様記述を計算機で処理して検証の自動化に繋げることを考慮する必要がある．現在，システムLSIなどの大規模ハードウェアの検証には，専用ハードウェアを使った高速シミュレーションや形式的検証など，様々な技術の併用が欠かせなくなっている．一つの仕様記述を様々な検証技術で共通して利用するために，それらの技術との親和性の高い仕様記述方式が求められている．

本文では，モジュール間インタフェースに適した仕様記述方式を提案し，仕様記述から仕様を監視する順序機械への変換によりポータビリティの高い検証モデルを実現するアプローチを紹介する．2節では現在の論理検証の形態とそこで必要となる入力情報について，3節では論理検証に必要な順序機械モデルを得るための仕様記述方式について述べる．4節では，実設計の検証への適用事例を紹介する．

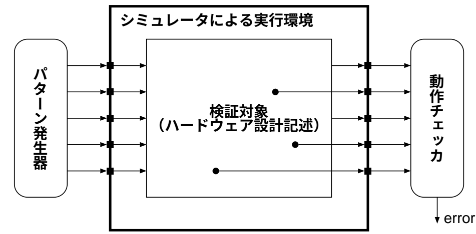

図 1: 論理シミュレーションの形態 (a)

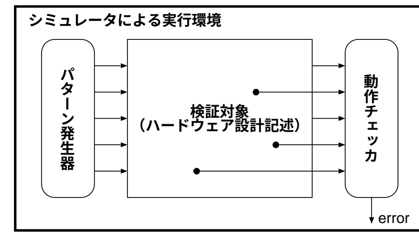

図 2: 論理シミュレーションの形態 (b)

2　論理検証の形態

2.1　論理シミュレーション

論理シミュレーションによる検証では，検証対象であるハードウェア設計記述に加えてハードウェアへの試験入力パターンを発生する機構とハードウェアの動作の正しさをチェックする機構が必要である（図 1，図 2）．パターン発生器の出力は検証対象の入力に接続され，動作チェッカは検証対象の入出力または内部信号を監視する．これらは，シミュレータの外部で実行される場合 (a)，検証対象と一体化してシミュレータ内で実行される場合 (b)，およびその組み合わせの場合がある．

(a) では，パターン発生器や動作チェッカとシミュレータの間での通信のオーバーヘッドがパフォーマンス上の問題になる場合がある．一方，シミュレータ内部に置く場合には，通信は大きな問題ではなくなる．一般に，ソフトウェアのみで構成された論理シミュレータの場合にはシミュレーションの実行速度と対比して上記の通信のオーバーヘッドはそれほど問題にならないが，FPGA や専用ハードウェア（アクセラレータ）を用いた論理シミュレーションでは通信のオーバーヘッドを無視できない場合が多いため，(b) の形態が好まれる．

(a) ではパターン発生器や動作チェッカの構成方法に自由度が高く，C/C++などのソフトウェア用言語を用いることも可能である．一方 (b) では，それらは使用するシ

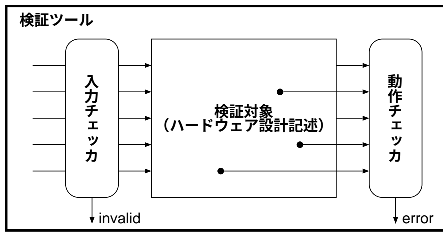

図 3: 形式的検証の形態

ミュレータ上で実行可能な形式でなければならない．多くの場合，それは検証対象と同じハードウェア記述言語（HDL）で順序機械を表現したものである．

2.2　形式的検証

形式的検証の技術を用いた検証（モデル検査やセミフォーマル検証など）では，個別の試験入力パターンではなく入力パターン集合全体を定義するものが必要になる．検証対象であるハードウェア設計記述に加えて必要なものは，ハードウェアへの入力の正しさをチェックする機構とハードウェアの動作の正しさをチェックする機構である（図 3）．入力チェッカは検証対象への入力を監視し，動作チェッカは検証対象の入出力または内部信号を監視する．検証ツールは，入力チェッカで異常入力が検出されないという条件の下で，検証対象の動作の正しさを調べる．一般に，形式的検証の技術では検証ツールと各チェッカの連携が密であり，それらは検証ツールの内部で処理される．

複数の検証ツールの併用が必要な現状では，仕様記述の共通化をはかることで，各ツール独自の形式による人手作業をできるだけ少なくしたい．検証対象と同じ順序機械を表現した HDL の形式であれば，どのような検証ツールでも共通して利用可能であろう．

3　順序機械に変換可能な仕様記述

3.1　仕様の記述形式

一般には，パターン発生器やチェッカの回路を HDL で直接記述することが難しくない場合も多い．しかしながら時系列を定義した仕様については，それを HDL で表現することはそれほど単純な作業ではない．特にインタフェース仕様では，時系列の仕様が多く現れる．

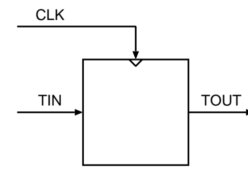

図 4: ハードウェア M1

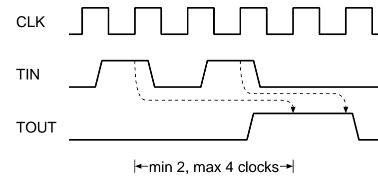

図 5: M1 の仕様の概略図

```
if (TIN==1) {
  wait(2,4);
  expect(TOUT==1);
}
```

図 6: M1 の仕様記述

図 4 のハードウェア M1 を例に，仕様記述の形式について考える．M1 はクロック信号 CLK，入力 TIN，出力 TOUT を持っており,「TIN に 1 が入力されると 2 から 4 サイクル後に TOUT から 1 が出力される」という仕様（図 5）があるする． 仕様上のある意味を持った一連の動作時系列を「トランザクション」と呼ぶ．ここでは，1 回の TIN の入力に対する 1 回の TOUT の出力までをトランザクションと見なす．M1 はこのトランザクションをパイプライン的に並列実行するハードウェアである．これを HDL で表現しようとすれば，パイプライン構造を明示的に実装する必要があることが予想できる．

そこで，より単純にトランザクションを記述する方法を考える．C 言語風の文法を使えば，トランザクション中心の仕様は図 6 のように表現することができる．ここで，`wait(2,4)` は 2 から 4 サイクルの範囲の時間経過を，`expect(TOUT==1)` は TOUT が 1 になることが期待されることを示している．トランザクション中心の記述によって，この種の仕様をより直感的に表現できる可能性があることがわかる．ただし，図 6 の文法では，数学的に厳密な意味付けがまだ十分ではない．次の節では，より数学的なモデルに基づいて厳密に仕様を定義する方法について検討する．

3.2　正規表現の利用

トランザクション中心の仕様記述を順序機械に変換する上で有効なのが，正規表現と有限オートマトンの技術である．

正規表現は入力記号の時系列の集合を表す．M1 における入力記号は，2 つの観測点 TIN，TOUT の値の組から成る (0,0)，(0,1)，(1,0)，(1,1) の 4 つである．基本となる正規表現は，下の 1，2 から出発して 3 を有限回繰り返すことによって定義される．

1. 「ε」は長さ 0 の時系列の集合を示す正規表現
2. 「a」（a は入力記号）は，記号 a だけから成る長さ 1 の時系列の集合を示す正規表現
3. R, S を正規表現とするとき
   - 「$R\,|\,S$」（R と S の和）は R と S の和集合を意味する正規表現
   - 「RS」（R と S の連結）は R の任意の要素と S の任意の要素を連結した時系列の集合を意味する正規表現
   - 「$R*$」は 0 個以上の R の連結を意味する正規表現（0 個の場合は ε）

正規表現は時系列の数学的な表現として優れている．我々はモデル検査用のプロパティ記述言語として正規表現を採用しており，これが実設計の検証において十分に実用的であることを確認している [1, 2]．また，設計資産の再利用を支援するために，インタフェース記述として正規表現ベースの言語を採用したアプローチも紹介されている [3, 4]．

M1 の例における不正なトランザクションの集合を正規表現で記述してみる．M1 の仕様に対する不正なトランザクションは，「TIN に 1 が入力された後，2 から 4 サイクルの間 TOUT から 1 が出力されない（~TOUT)」という時系列である．これは次のように表現できる．

.*[TIN].[~TOUT]{3}.*　　(1)

ここでは以下の簡略記法を用いた．

- 「.」は長さ 1 の任意の時系列の集合
- 「[f]」は条件式 f を満たす長さ 1 の時系列の集合
- 「$R\{n\}$」は n 個の R の連結

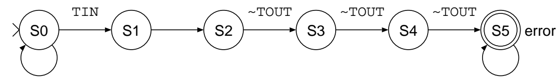

図 7: M1 の不正パターンを検出する非決定性有限オートマトン

```
module checker(CLK, TIN, TOUT, error);
  input CLK, TIN, TOUT;
  output error;
  reg S0, S1, S2, S3, S4, S5;

  assign error = S5;

  initial begin
    S0 = 1;
    S1 = 0;
    S2 = 0;
    S3 = 0;
    S4 = 0;
    S5 = 0;
  end

  always @(posedge CLK) begin
    S0 <= S0;
    S1 <= S0 & TIN;
    S2 <= S1;
    S3 <= S2 & ~TOUT;
    S4 <= S3 & ~TOUT;
    S5 <= S4 & ~TOUT | S5;
  end
endmodule
```

図 8: M1 の不正パターンを検出する順序機械の HDL 記述

すなわち，. は ((0,0)|(0,1)|(1,0)|(1,1)) と等価，[TIN] は ((1,0)|(1,1)) と等価，[~TOUT]{3} は ([~TOUT][~TOUT][~TOUT]) と等価である．

正規表現は有限オートマトンに対応している．例えば正規表現 (1) は図 7 の非決定性有限オートマトンに対応している．図中でラベルのない枝（$0 \to 0$ と $1 \to 2$）は任意の条件で遷移できることを示している．開始状態 0 から出発して受理状態 5 に到達すると，そこまでの時系列は不正パターンである．

非決定性有限オートマトンを HDL で実装する簡単な方法の一つは，n 個の非決定的状態に対して n ビットのレジスタを用意し，各ビットがオートマトン状態の複数の可能性を示すように論理を構成することである．図 7 の非決定性有限オートマトンを Verilog-HDL で実装した例を図 8 に示す．この HDL 記述は単純な同期回路であり，ほとんどの論理シミュレータ／検証ツールと組み合わせて実行可能であろう．

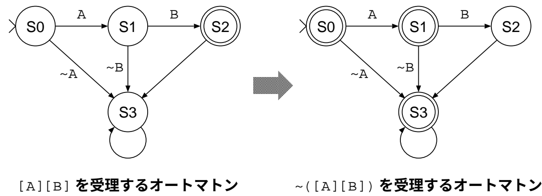

図 9: 補集合の有限オートマトンの構成

3.3 PSE による仕様記述

我々は前節で紹介した正規表現の記法に以下の記法を加え，それらを用いたハードウェア動作の表現を PSE（Path Set Expression）と呼んでいる．

- 「R?」は R または ε
- 「R+」は 1 個以上の R の連結
- 「$R\{n,\}$」は n 個以上の R の連結
- 「$R\{n,m\}$」は n 個以上 m 個以下の R の連結
- 「<R>」は後ろに長さ 0 以上の何らかの時系列を連結することによって R の要素になり得る時系列の集合
- 「~R」は R に含まれない全ての時系列の集合
- 「R&S」は R にも S にも含まれる時系列の集合

<R> に対応する有限オートマトンは，R の有限オートマトンにおいて受理状態へのパスを持つ状態全てを受理状態とすることで得られる．~R に対応する有限オートマトンは，まず R の完全な決定性有限オートマトンを構成し，受理状態と非受理状態を交換することで得られる（図 9）．R&S は ~(~R|~S) と等価である．

仕様を表現する際には以下の 4 つの形式を用いる．

- 静的な条件を論理式で記述する *static* 形式
- 正常パターンの PSE を記述する *normal* 形式
- 不正パターンの PSE を記述する *error* 形式
- 条件を示す PSE と条件成立後の動作を規定する PSE を記述する *if-then* 形式

if-then 形式「*if* R *then* S」は，R に含まれる時系列が発生した後には，必ずそれに続く時刻から S に含まれる何らかの時系列が発生することを意味する．言い換えれば，「*if* R *then* S」は R に含まれる時系列が発生した後，それに続いて S に含まれる時系列が（過去にも未来にも）発生しない状況を不正パターンとする仕様である．この不正パターンは「.* R ~< S .*>」と表現できる．この規則を用いれば，任意の *if-then* 形式を不正パターンの PSE に変換することができる．不正パターンの PSE からは，前述のように有限オートマトンを経て順序機械の HDL 記述を得ることが可能である．

if-then 形式の導入により，柔軟で直感的なトランザクション記述が可能になった．M1 の仕様は，*if-then* 形式で次のように記述することができる．

$$\textit{if}\ \texttt{[TIN]}\ \textit{then}\ \texttt{.\{1,3\}[TOUT]} \qquad (2)$$

この仕様からは，上記の規則を用いて図 8 と同様の順序機械を得ることができる．

4 実設計への適用例

我々は PSE による仕様記述を HDL 形式のチェッカに変換するツール「PSE2HDL」を開発し，実設計の検証に利用している．本説では，PSE2HDL を組み込み用プロセッサコアの検証モデル作成に利用した事例について紹介する．

この事例では，命令フェッチ部，演算実行部，キャッシュ，レジスタファイルなどのモジュールが比較的密に協調して動作している．そのようにモジュール間の依存関係が強い部分では，直接 HDL を使って疑似モデル（Pseudo）を記述する方が，PSE だけで入力制約を記述するよりも易しい場合がある．その場合でも PSE 記述を併用することで，全体としての検証モデル作成の簡単化や二重チェック（Pseudo のチェックを含む）による信頼性向上を実現することができる．

演算実行モジュールの検証のために作成した検証モデルの構成を図 10 に示す．検証対象（演算実行モジュール）は，命令フェッチ部から供給される命令を処理する．命令コードには組み合わせ禁止のビットパターンが存在する．その条件を PSE で記述し，PSE2HDL を使ってチェッカに変換した．演算実行時にはパイプラインの競合による命令供給の一時停止などが発生する．そのような状況を模擬するフェッチ部の Pseudo は，直接 HDL で記述した．その他の外部環境であるキャッシュ，レジスタファイルなどの概略動作を模擬する Pseudo も HDL で記述し，検証対象に接続した．検証対象と各 Pseudo の間のインタフェース仕様については PSE で記述した．PSE2HDL

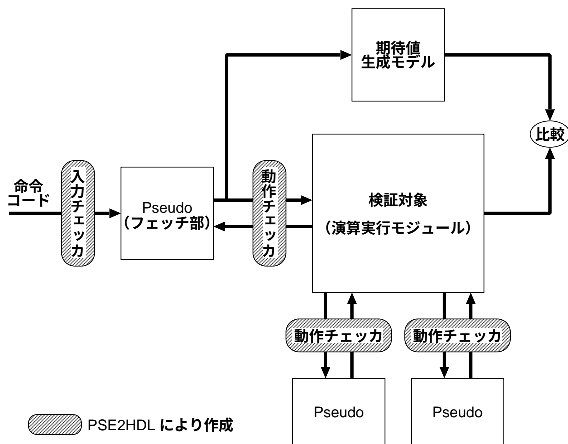

図 10: 検証モデルの構成

を使ってチェッカに変換し，検証対象と各 Pseudo の間の通信が正しく行われているかどうかを監視させた．検証対象の最終的な動作は，期待値を生成する動作モデルとの信号値比較で確認した．

PSE によるインタフェース仕様記述の例を示す．演算実行モジュールは wait 信号がアサートされると一時停止する構成になっており，ここでは wait 信号がアサートされないサイクルを実行サイクルと呼ぶ．

1. 実行サイクルで illegal 信号がアサートされると，次のサイクルに必ず cancel 信号がアサートされる．

   if `[illegal&~wait]` *then* `[cancel]`

2. 実行サイクルで inst_ld 信号がアサートされると，その後 2 実行サイクルを消費した後に rd_valid 信号がアサートされる．

   if `[inst_ld&~wait]([wait]*[~wait]){2}`
   then `[rd_valid]`

3. wait は最長でも 3 サイクルしか持続しない．

   error `[wait]{4}`

全体では 116 件のインタフェース仕様が記述された．タイプ別に分類すると，*static* 形式が 54 件，*normal* 形式が 44 件，*error* 形式が 2 件，*if-then* 形式が 16 件であった．これらのインタフェース仕様から PSE2HDL によって自動生成されたチェッカの HDL 記述量と，検証対象および Pseudo の HDL 記述量を比較すると次のようになる．

検証対象	14,320 行
Pseudo 合計	1,518 行
チェッカ合計	1,818 行

検証モデル全体をセミフォーマル検証ツールに与え，入力チェッカが異常を検出しないことを条件に入力（命令コード）を変化させるように指定することで，演算実行モジュールの論理検証を実現することができた．また，演算実行モジュール内で命令デコードを担当するサブモジュールについては，モデル検査を実施することにも成功した．その際にはフェッチ部との間のインタフェース仕様の一部を再利用し，Pseudo 等その他の記述は不要であった．

5 おわりに

本文では，PSE によるトランザクション中心の仕様記述と，実設計のモジュール検証におけるその役割について報告した．我々は，PSE から対応する順序機械を表現した HDL 記述への変換ツールを実現し，一つの仕様記述を様々な検証技術で共通して利用する枠組みを提案している．トランザクション中心の仕様記述は HDL 的な記述スタイルを補完するものとして，特にインタフェース仕様を記述する工数を削減する上で実用性があることが確認された．また，この方法では自然に設計記述とは異なる角度からの仕様記述となるため，仕様記述中に設計と同じ誤りを入れてしまう危険性を避ける効果も期待できる．我々は，これらの効果をより大きいものとするために，現実の仕様タイプについてより多く調査し，仕様記述方式を改善していく方針である．

参考文献

[1] H. Iwashita and T. Nakata, “Forward Model Checking Techniques Oriented to Buggy Designs,” in *Proc. ICCAD-97*, pp. 400–404, 1997.

[2] H. Iwashita and T. Nakata, “Efficient Forward Model Checking Algorithm for ω-Regular Properties,” in *IEICE Trans. Fundamentals*, E82-A(11), pp. 2448–2454, 1999.

[3] Kei Suzuki, Kouji Ara, and Kazuo Yano, “OwL: An Interface Description Language for IP Reuse,” in *CICC99*, pp. 403–406, 1999.

[4] 荒 宏視，鈴木 敬，矢野 和男,「IP 再利用のためのインタフェース記述言語 O_wL の提案」，DA シンポジウム ’99，pp. 15–20，1999.