設計自動化 47－1
（1989. 5. 23）

ルールベース論理設計ルール検証プログラム

小迫 靖志　大坪 千恵　荻原 拓治　村井 真一

三菱電機　カスタムＬＳＩ設計技術開発センター

設計された論理回路が試験容易化設計ルールに従い正しく設計されていることを検証するプログラムにルールベース手法を導入し、設計ルールの保守・管理の容易化を図った。この検証プログラムの特徴は、設計ルールだけでなく、検証用記号シミュレータで使われる真理値関数もルールベース化している点と、予め定義しておいたプリミティブ素子と記号信号値を用いて、設計ルールと真理値関数を記述できる専用言語をサポートしている点にある。これにより、設計ルールの追加・変更を容易に行えるため、設計ルールの保守・管理作業を短縮することができる。

RULE-BASED LOGIC DESGIN RULE CHECK PROGRAM

Yasushi Koseko, Chie Otsubo, Takuji Ogihara and Shinichi Murai

ASIC Design Engineering Center, Mitsubishi Electric Corporation
5-1-1 Ohfuna, Kamakura, 247 JAPAN

This paper describes a logic design rule check program which adopts a rule-based technique in order to manage design rules easily. It treats not only the design rules as a rule base, but also operation of primitives which are referred by a symbolic simulator. Moreover, a rule-based description language is prepared, in which design rule managers can easily describe the design rules and the operation of primitives by using the predefined primitives and symbolic signal values.

1 はじめに

半導体技術の進歩にともなう LSIの高集積化により LSI等論理装置の試験に関わる費用が急増している。この問題を解決するために様々な試験容易化設計手法が提案されており、特にスキャン設計[1][2]は広く知られている。これを適用するには論理回路が設計ルールに従い正しく設計されていることを確かめる必要があり、そのチェックに設計ルール検証プログラムが使用される。設計ルールは当初スキャン設計を主とする試験容易化のためのものであったが、設計者間で取り決めた設計指針が設計ルールに加えられたり、プロジェクトごとに設計ルールが設定されることが多くなっている。このため設計ルールを柔軟に取り扱う必要が生じている。

しかしながら、従来の論理設計ルール検証プログラム[3]は設計ルールがプログラム内に埋め込まれているため、設計ルール更新のたびにプログラムの改修作業が生じた。その更新の影響が検証対象となるプリミティブ素子や、検証用シミュレーションの信号値・真理値関数の改修にまで及ぶ可能性もあるため、たとえわずかな設計ルールの変更であってもプログラム全体にわたる改修や最悪の場合には作り直しの可能性もあった。この状況は検証プログラムの維持・管理の面で好ましくないだけでなく、論理装置開発の期間短縮を阻害する原因にもなっていた。

そこで我々は、この問題を解決するために、検証プログラム中に埋め込まれていた設計ルールと[4][5]、真理値関数をルールベースとして独立させた。これにより、設計ルールの変更があっても、プログラム本体を改修することなくルールベースを書き換えるだけで検証プログラムの更新が可能となった。

本稿では、ルールベース手法を全面的に採用することでプログラム保守・管理の容易化を図った設計ルール検証プログラムについて報告する。

2 検証プログラムの特徴

以下に本検証プログラムの特徴を示す。

(1) 検証用シミュレーションで信号値として記号値を採用している。
(2) プリミティブ素子と信号値を外部で定義することができる。
(3) 真理値関数と設計ルールをルールベース化している。
(4) (2)で定義したプリミティブ素子と記号値を使って真理値関数と設計ルールを記述できる専用言語をサポートしている。

3．プログラム構成

本検証プログラムは次の4つの部分からなる(図1)。

(1) シェル
(2) コマンド
(3) 内部メモリ
(4) 外部ファイル

シェルはユーザからの命令を解釈し、その命令に適したコマンドをコールする。コールされたコマンドは内部メモリや外部ファイルをアクセスすることによって、その役割を果たす。このステップを繰り返して設計ルール検証処理を行う。

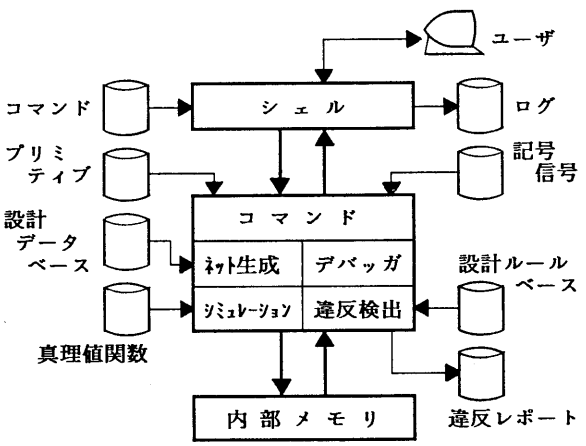


図1 プログラム構成

組み込まれている主なコマンドは4グループに分けられる。

①回路ネット生成
②記号シミュレーション
③設計ルール違反検出
④デバッガ

①は回路データベースから回路データを入力し内部メモリに回路ネットを構築する処理である。②は設計ルール違反検出で使われる信号値をシミュレーションにより求め内部メモリに蓄える処理である。シミュレーションは外部入力から外部出力へ信号値を伝搬することにより行われる。回路内の各素子の出力値はその入力信号値と真理値関数から決定される。③は①で生成した回路ネットと②で得た信号値を調べて設計ルールに違反しているところがないかチェックする処理である。設計ルールはルール違反の型と違反内容の説明文からなり、その違反の型が回路内の違反部分とマッチしたときその違反内容が出力される。④は内部メモリのダンプや回路ネットのトレース等のデバッグ・コマンドである。内部メモリは全てのコマンドからアクセス可能で、回路ネットや信号値、設計ルール違反情報などを蓄える。

4 プリミティブ素子及び記号信号

ここでは設計ルール検証プログラムが認識できなければならない基礎的な知識つまりプリミティブ素子及び記号信号に関するデータをプログラムに与える方法について述べる。

4.1 プリミティブ素子

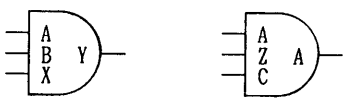
設計ルール検証プログラムは予めプリミティブ素子に関するデータをプリミティブ素子指定ファイルから入力し、回路ネットを生成するときや検証用シミュレーションを実行するときなどにこれを参照する。このファイルには各プリミティブ素子の

(1) タイプ名
(2) ピン名
(3) ピン属性

が記述される。図2はプリミティブ素子指定ファイルの例である。このファイルではタイプ名がAND、入力ピン名が(A-X)あるいは(0-9)、出力ピン名がYで、全てのピンがデータピンであるプリミティブ素子を指定している。図3の(a)にこの指定ファイルの通りにつくられた正しいANDのブロック図を、また(b)には間違ったブロック図を示す。

```
AND()
{
   pin(input)
   {
      [A-X] data ;
      [0-9] data ;
   }
   pin(output)
   {
      Y    data ;
   }
}
```

図2　プリミティブ素子指定ファイル



(a) 正しい　　(b) 誤り

図3　ANDのブロック図

4.2 信号値

検証用シミュレーションで使われる信号値は図4に示すように記号列で表される。記号列は記号の並びであり、記号は文字の並び、つまり文字列である。図5では記号信号値の例を示している。

記号列の中には信号値として正しいものと正しくないものがある。これを識別する方法とし記号列の組立方の違いを利用する方法がある。記号列の組立方を表したものを記号列の構成ルールと呼

び、この構成ルールに従って作られた記号列を正しい信号値、それ以外の記号列を不正な信号値、と区別することができる。

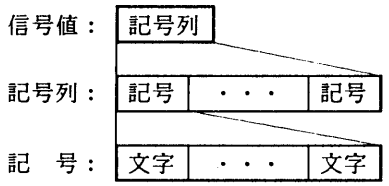


図4　記号値

・固定値　　HIGH　LOW

・データ　　DATA

・クロック*　C01U　C02U　C03U　C04U
　　C01D　C02D　C03D　C04D

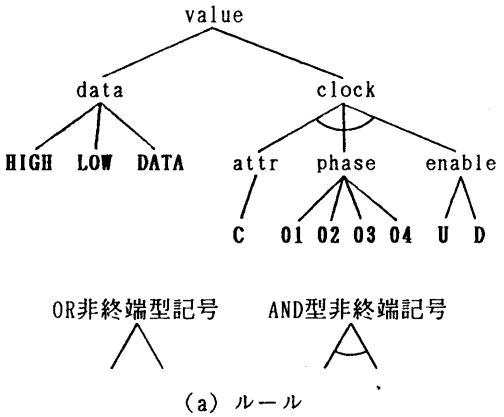
＊ Cはクロックを、01-04は相を、
U/Dは有意値を表す。

図5　記号信号値の例

信号値の構成ルールは木構造により容易に表現できる。図6は図5の信号値に対する構成ルールを木で表現した例である。葉にあたる記号を終端記号、その他の節点を非終端記号といい、特に根にあたる記号を開始記号と呼ぶ。記号列を構成する記号はすべて終端記号であり、非終端記号は構成ルールを表現するための中間的な記号として使われている。

非終端記号はその枝の分かれ方により2つのタイプに分けられる。一つはいずれか一つの枝を満足すればよい OR 型非終端記号で、他の一つは全ての枝を満足しなければならない AND 型非終端記号である。

この木を根から葉に向かって辿ることによって、全ての信号値を生成することができるし、また与えられた記号列が正しい信号値かどうかを判定することもできる。



(a) ルール

非終端記号 ： value data clock
　　attr phase enable
終端記号 ： **HIGH LOW DATA**
　　C 01 02 03 04 U D

(b) 記号

図6　記号信号の構成ルール

次に記号列 C03U がこの構成ルールを満足する記号列かどうか、つまり正しい記号値かどうか調べてみる（図7）。

まず、構成ルールの開始記号 value から調べ始める。 value は OR 型の非終端記号なので2つの枝のうちどちらか一方が成功すればよい。左側の枝 data を調べてみると、その下に終端記号（HIGH、LOW、DATA）があるが、いずれの記号も C03U にマッチしない。右側の AND 型非終端記号 clock を調べてみると、その下には3つの非終端記号（attr、phase、enable）があり、それぞれ C03U の（C、03、U）にマッチする。したがって、記号列 C03U が正しい信号値であることが判る。

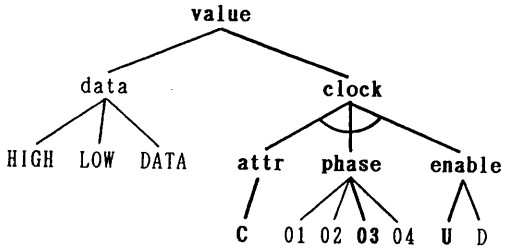


図7　記号値C03U の判定

図8に別の例を示す。この図から記号列 C02H が信号値として正しくないことが判る。C02H の H は構成ルールのどこにも定義されていない。

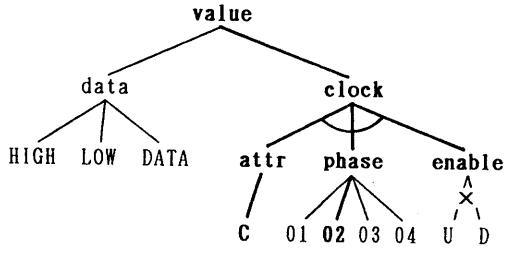


図8　記号値C02H の判定

設計ルール検証プログラムに信号値の構成ルールを与える書式は ＢＮＦ (Backus Naur Form) に似た形式を用いる。図9に図6の構成ルールの記述例を示す。構成ルールの記述は次の３つの部分から成る。

（1）記号の構成文字の定義
（2）記号の定義
（3）記号列の定義

（1）では記号を構成する文字の集合を定義する。プログラミング言語のデータ型の宣言にあたる。（2）では開始記号以外の記号を定義する。終端記号は式の右辺に現れ、非終端記号は左辺に現れる。

（3）では開始記号及びその他の非終端記号の関係を指定する。

```
/*  記号値の構成文字の定義  */
  typedef TypeAlpha [A-Z] ;
  typedef TypeDigit [0-9] ;
  typedef TypeEmpty []    ;
/*  記号値の定義  */
  TypeAlpha data  = { HIGH,LOW,DATA } ;
  TypeEmpty clock ;
  TypeAlpha attr  = { C } ;
  TypeDigit phase = { 01,02,03,04 } ;
  TypeAlpha enable= { U,D } ;
/*  記号列の定義  */
  Syntax(value)
  {
     value : data
           | clock
           ;
     clock : attr phase enable
           ;
  }
```

図9　信号値構成ルールの記述例

5 記述言語

本検証プログラムでは予め定義したプリミティブ素子及び記号信号値を使って設計ルールや真理値関数を記述できるよう専用の言語が用意されている。この言語はＣ言語のサブセットに＃関数という組み込み関数を加えて作られている。＃関数にはプリミティブ素子・記号信号値を操作するための関数や、回路ネットなどのテーブルへのアクセスを容易にするための関数が含まれている。＃関数の例を表１に示す。

記述言語 ＝ Ｃ言語 ＋ ＃関数
↓ プリコンパイル
Ｃ言語

表１　＃関数の例

＃関数	機能
#in()	入力ピンテーブルへのポインタを得る。
#out()	出力ピンテーブルへのポインタを得る。
#get(v,n)	信号値ｖから非終端記号ｎにあたる部分の記号を取り出す。
#put(v,n,s)	信号値ｖの非終端記号ｎにあたる部分に記号ｓを挿入する。
#match(v,n,s)	信号値ｖの非終端記号ｎにあたる部分が記号ｓに等しいとき真を、等しくないとき偽を返す。
#compare(v,w)	信号値ｖと信号値ｗが等しいとき真を、等しくないとき偽を返す。
#reset(v)	信号値ｖをリセットする。つまり、ｖに空文字列をセットする。

専用言語で記述された設計ルール及び真理値関数は専用のプリコンパイラによって一度Ｃ言語に変換された後、設計ルール検証プログラム本体と共にコンパイル／リンクされる。プリコンパイラは unix のプログラム開発ツールである字句解析ツール lex (lexical analyzer) とパーザ生成ツール yacc (yet another compiler-compiler) を併用して作られている。

５．１ 真理値関数

専用記述言語を用いて真理値関数を記述する方法をｎ入力ANDプリミティブ素子の例で説明する（図１０）。ここで採用している信号値は図９で定義した記号信号値である。

図中（1）ではプリミティブ素子の名前を指定している。（2）では変数の宣言を行っている。変数名は$で始まる英数字からなる文字列である。（3）では２入力and関数を繰り返し用いてｎ入力AND の出力値を求めてる。変数$AND は中間変数として使われており、最終的にｎ入力AND の出力値を蓄える。組み込み関数 #in はｎ入力AND の入力ピンテーブルへのポインタを得る組み込み関数である。ピンテーブルには各ピンの信号値が納められている。（4）では $AND に蓄えられているｎ入力AND の出力値を出力ピンテーブルに書き込んでいる。出力ピンテーブルへのポインタは組み込み関数 #out により得ることができる。

```
AND()    ----------------------------(1)
{
   TypePin *$i,*$o, $AND;  ----------(2)

   $AND=(*#in);
   for($i=#in+1;(*$i)!=NULL;$i++)   -(3)
      $AND=and(*$i,$AND);

   for($o=#out ;(*$o)!=NULL;$o++)   -(4)
      *$o=$AND;

   return(0);
}
```

図１０　真理値関数の記述例

次にｎ入力AND の出力値を求める際に繰り返し使われている２入力and の真理値関数を図１１（a）に示す。また参考のためにこれを真理値表

で表したものを図１１（b）に示す。（a）と（b）両図において同一番号の部分は同じ内容を表している。

```
and($a,$b)
TypePin $a,$b;
{

   switch(#get($a,data)) {          ----(1)
   case "HIGH" : return($b);
   case "LOW"  : return($a);
   }

   switch(#get($b,data)) {          ----(2)
   case "HIGH" : return($a);
   case "LOW"  : return($b);
   }

   if( #match($a,attr  ,"C") &&     --(3)
       #match($a,enable,"U") &&
       #match($b,data,"DATA") )
      return($a);
   if( #match($b,attr  ,"C") &&
       #match($b,enable,"U") &&
       #match($a,data,"DATA") )
      return($b);
   if( #match($a,attr,"C") &&
       #match($b,attr,"C") &&
       #compare($a,$b)       )
      return($a);

   #reset($b);                    --------(4)
   #put($b,data,"DATA");
   return($b);

}
```

図１１(a)　２入力andの真理値関数

$b＼$a	HIGH	LOW	DATA	CaxU	CayD
HIGH	HIGH	LOW	DATA	CaxU	CayD
LOW	LOW	LOW	LOW	LOW	LOW
DATA	DATA	LOW	DATA	CaxU	DATA
CbxU	CbxU	LOW	CbxU	Vx	DATA
CbyD	CbyD	LOW	DATA	DATA	Vy

(2)　(4)　(1)　(4)　(3)　(4)

Vx = CaxU(ax＝bx), DATA(ax≠bx).
Vy = CayD(ay＝by), DATA(ay≠by).

(3)　(4)

図１１(b)　２入力andの真理値表

５．２ 設計ルール

設計ルールの記述方法は真理値関数の場合によく似ている。説明のために『Ｄラッチのデータ入力ピン（Dピン）にクロックを印加してはならない。』という設計ルールを取り上げる。この設計ルールを専用記述言語を用いて記述すると図１２（a）のようになる。図中（1）では設計ルール名を指定している。（2）はDピンにクロック信号が伝搬するＤラッチを回路の中から捜し出し、もしあればルール違反メッセージを出力する部分である。#CONNECT は組み込み関数で回路内の全接続ポイントにおいて条件（この場合、素子タイプが DL でそのピンが D の接続ポイント）にマッチすると真を返し、マッチしなければ偽を返す関数である。#match は信号値用の比較関数である。

図１２（b）の回路は（a）の設計ルールに違反している回路の例である。素子名 D-latch の D ピンに CO2U というクロックが伝搬しているため違反である。

```
TypeRule ansynchronous()  ----------(1)
{
   while( #CONNECT:$dl T=DL; P=DL.D; )
   {
      if( #match($dl->value,attr,"C") )
      {
         message("A clock .. to '%s'.",
                  instance($dl->elm));
      }                                  /
   }                                   (2)
}
```

図１２(a)　設計ルール記述例

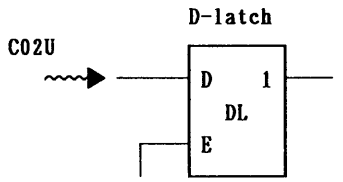


図１２(b)　設計ルール違反例

6 実行結果

現在、スキャン設計ルールの組み込み作業を行っているところである。すでにプリミティブ素子、記号信号ルール、真理値関数の作成は終えており、設計ルールも基本的なルールを約４０組み込んだところである。組み込んだ設計ルールには回路を同期化するためのルールやスキャン動作を保証するためのルール等が含まれてる。これらの設計ルールのチェックを３つのサンプル回路について実施した。その結果を表２に示す。

表２　実行結果

回路	素子数	ＣＰＵ(秒)
A	１５９	２．６
B	６２１	４．２
C	３３７９	２３．７

４MIPSマシンを使用

7 おわりに

本報告では、ルールベース手法を積極的に採用した設計ルール検証プログラムについて述べた。本検証プログラムの大きな特徴は、設計ルールだけでなく、検証用記号シミュレータで使われる真理値関数もルールベース化している点と、予め定義しておいたプリミティブ素子と記号信号値を用いて、設計ルールと真理値関数を記述できる専用の言語を提供している点である。これにより、設計ルールの追加・変更に対し柔軟な対応ができるため、設計ルールの保守・管理に関わる作業の短縮が可能となる。

今後は、スキャン設計ルールの組み込みを完成させるとともに、設計ルール違反原因の指摘や違反回路の自動変換機能もサポートしてゆく予定である。

参考文献

[1] Eichelberger,E.B. and T.W.Williams, "A Logic Design Structure for LSI Testing," proc. 14th DA Conference pp462-468

[2] Funatsu,S. ,et al., "Design Digital Circuits with Easily Testable Considerations," 1978 Test Conference

[3] Muroi,K. ,et al., "A HIERARCHICAL LOGIC DESIGN CHECK PROGRAM FOR SCAN DESIGN CIRCUITS," 1986 ICCAD'86 pp106-109

[4] Horstmann,P.W. and E.P.Stabler, "Computer Aided Design Using Logic Programming," 1984 DA Conference pp144-151

[5] Kyushik Son, "RULE BASED TESTABILITY CHECKER AND TEST GENERATION," 1985 Test Conference