設計自動化 36－1
（1978. 2. 17）

ハードウェアの機能テスト支援システムの考察

小林 一夫　若林 春夫
NTT電気通信研究所

本論文では、ハードウェアの機能仕様を表わす記述から機能テストデータを自動生成するシステムの実現法とシステムの試用結果の考察を述べている。機能仕様はハードウェアへの入力データがどのような順序で加工されて出力されるかを規定する。これを形式的にかつ見通しよく記述可能とするため、ハードウェアの特長を考慮した専用の記述言語を開発した。機能テストデータの生成では、この記述をもとに入力データとそれに対して期待される出力データを対応付ける。このとき、規定以外の入力による出力も確認できるように生成手順を設定している。このシステムをいくつかのハードウェア仕様に適用した結果、人手設計に近いテストデータが生成でき、その量は記述量の２乗以下であることが確認できた。

FUNCTIONAL TEST DATA GENERATION FOR LOGIC DESIGN VERIFICATION

Kazuo KOBAYASHI　Haruo WAKABAYASHI

NTT Electrical Communication Laboratories,Musashino-Shi,Tokyo 180 Japan

This paper presents a test design support system that can generate functional test data automatically from descriptions of hardware functional specifications. A functional specification provides information regarding how data input to the hardware are processed and which outputs they are sent to. To describe the specification formally, we have developed a new description language taking the characteristics of hardware functions in consideration. This system uses an algorithm that can generate two types of test data from the descriptions. Those test data can check the output expected not only from the specified input but also from an unspecified input. Some preliminary experiments show that this system can automatically generate almost the same amount of test data as can be produced manually, and that the test data require less than the second power of the number of descriptions.

1 はじめに

情報社会の進展と部品技術の向上とともに、ハードウェアが大規模化する方向にあるため、設計の自動化をめざした研究が各所で進められている。しかし、現在の研究の主な対象は機能設計以降の自動化であり、ハードウェアの持つべき機能の規定(”機能仕様”)の決定から機能設計までは人手に頼っているため、依然として設計誤りが生じる原因となっている。高集積化されたハードウェアでは、設計誤りを製造後に修正するのが難しいので製造前に取り除く必要があり、多くの設計工数を要している。そのため、テストの精細化とテスト作業の省力化が重要な課題となってきている。

このような背景のもとで、上記の設計自動化にあわせてテスト支援手法についても、テストプログラム専用の高水準言語[1)-3)]、論理検証の自動化手法[4)-6)]等の研究がなされている。

しかし、いずれの方法でもテストデータは人手で作成することを前提としているため、テストデータに人為的誤りが混入する危険が問題として残されている。

この問題を解決するために、ハードウェアの機能仕様を表わす記述からテストデータを自動生成するシステムを開発した[7)-9)]。本論文では、このシステムを実現する機能仕様の記述言語とテストデータ生成法を述べた後、その試行結果をもとに、機能仕様記述言語の記述性、自動生成されたテストデータの量等を評価する。

2 機能テスト支援

ハードウェアの設計では、機能仕様や性能の規定のような要求条件と使用できる部品技術等の設計制約をもとにして実際の回路が作られる(図1)。同じ要求条件が与えられても設計制約が異なると、設計される回路構成が異なる。しかし、設計が正しければ、いずれの回路構成ももとの要求条件を満たしている。この考えにもとづいて、設計の正当性を回路構成に依存せずに保証するために、設計された回路と機能仕様との照合(すなわち機能テスト)が設計過程で行われる(図2)。この機能テストを効率化、高精度化するために、テストデータの自動生成を以下の方法で実現する。

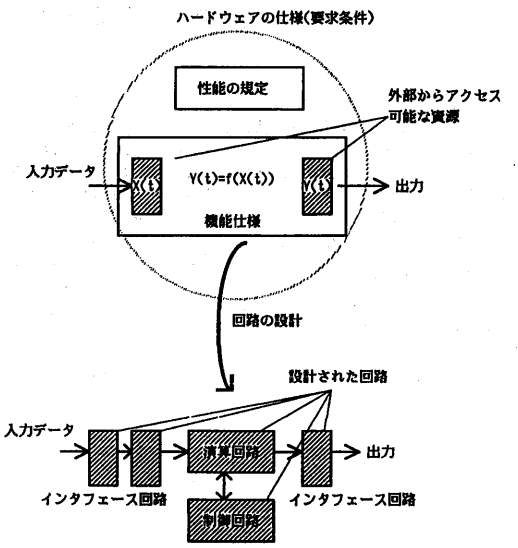


図1 ハードウェアの仕様と実現回路

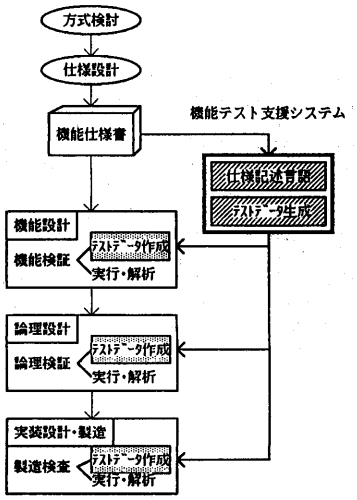


図2 機能テスト支援

3 仕様記述言語

3．1 記述モデル

機能仕様では、対象とするハードウェアへの入力データがどのような順序で加工されて出力されるかを記述する必要がある。形式的な記述を可能とするために、次のようにハードウェアをモデル化する。すなわち、ハードウェアは、外部(他のﾊｰﾄﾞｳｪｱまたはこれを利用するｿﾌﾄｳｪｱ)との情報交換の構造を規定するインタフェース資源(以下では資源)と命令機能、外部装置制御機能などのような関連の強い機能同士をまとめたプロセスとから構成される。入力データの加工と出力を、このプロセスにおけるデータの演算および資源へのデータ転送として表わす。プロセス間のデータ転送動作は、他方のプロセスを呼び出すことで表す(図３)。

3．2 言語の構成法

大規模なハードウェアの機能仕様では上記のデータ転送動作を基本要素とする記述が数kステップから数十kステップになる。この記述・修正を容易化するには記述が簡明であること、判読性が良いこと等が求められる。それに対しては、記述対象をいくつかの範ちゅうに整理でき、それぞれに適した記法が使える方法が有効と考えられ、ここでは以下とする。

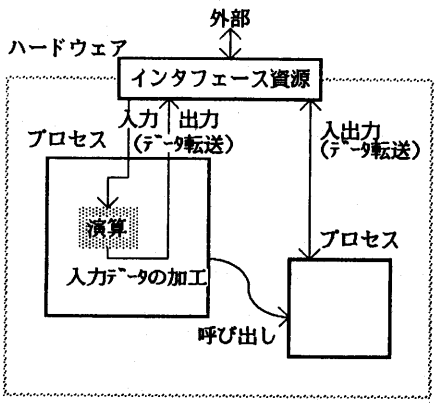


図3 記述モデル

(1)機能定義と属性定義

１つのプロセスとして表わされる機能の集合は、対象のハードウェアが果たすべき標準的な働きである典型的機能と例外処理のような補足的機能とに分けることができる。記述言語に機能定義部と属性定義部を設けて、この２つの機能を陽に書き分けられるようにする。

(2)手続き型記述と宣言型記述

上記の典型的機能の中には、プロセッサの命令の機能のように、命令のフェッチ、命令の実行、次アドレスの計算等の類型的な動作順序で表せる機能が多数存在する。これに着目して、動作順序を記述する手続き型の記述に、記述順が自由な宣言型の記述法を取り入れ、動作順序と動作内容の詳細を独立に記述できるようにする。

(3)制御機構の図的記述

動作順序は、if_then,ループ等の制御機構で表わせるが、それらを視覚的に表現できるHCPチャート形式の図的記述を導入する(図４)。

3．3 記述・修正法

以上の言語には図的記述とテキスト形式の記述の混在しているため、次の

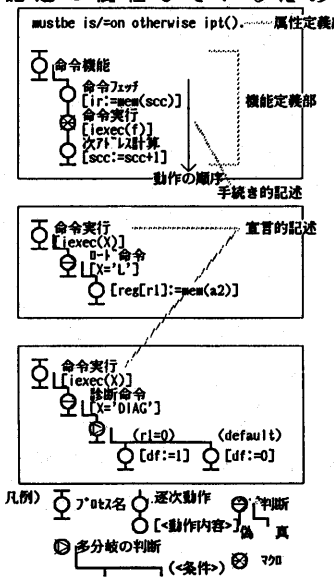


図4 仕様記述言語の構成

機能の専用の図的エディタ(図５)を用いて、記述・修正の容易化を図る。
(1)記述、編集用のコマンドと図記号の指定をメニュー形式で可能とする。
(2)記述、編集作業を「操作の指示(追加、移動等)」、「対象(図記号等)の指示」、「編集箇所の指示」の３つの基本操作に整理し、各基本操作の繰り返しが可能なコマンド処理方式をとる。

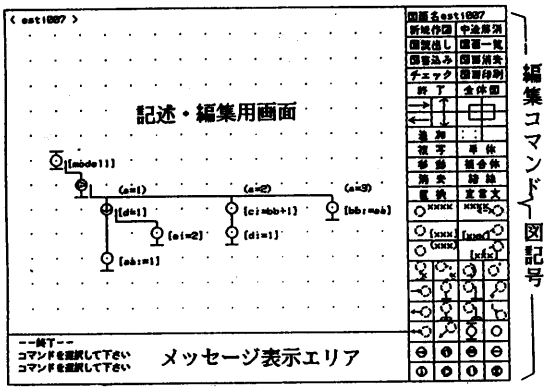


図5　仕様記述エディタの表示画面構成

４　テストデータ生成法

機能テストデータを自動生成するために、以下の手法で入力データとそれに対して期待される出力(入出力の"因果関係")を対応付ける。

４．１　入力と出力の判別

データの入出力は資源へのデータ転送(入力ﾃﾞｰﾀの加工もﾃﾞｰﾀ転送の一部として一括して扱う)として記述されるので、入力データと出力の判別が最初に必要となる。判別を機械的に行えるようにするため、与えられた仕様記述を展開して、①資源へのデータの転送、②その資源の値の判断、③判断にもとづく次のデータ転送、の３つの基本的な動作の繰り返しに定型化する。基本的動作の繰り返しの開始から終了までを１つの"経路"とし、１つの経路上の'判断'に使われる資源へのデータ転送をデータ入力、その結果のデータ転送を出力とする。

４．２　依存関係の探索

次に経路間の関係を考慮する。'判断'に使われる資源へのデータ転送と該当の'判断'が異なる経路に属する場合、一方の経路に属する動作が他方の経路の動作順序を決めるので、それらを"依存関係を持つ経路"と呼ぶ(図６)。さらに'判断'に使われる資源へのデータ転送が属する経路に対して、'判断'の属する経路を"下位の経路(逆は上位の経路)"と呼ぶ。

依存関係を持つ経路があれば、その経路同士を連結して新たな経路を作り、そこからテストデータを作る必要がある。それに対しては以下の方法をとる。

(7)依存関係は、図７に示すように連鎖的につながるので、テストデータの生成時間は、この連鎖を辿る手順に左右される。ここでは、生成を短時間で行うため、一階層下の経路のみ探索する手順をとる。すなわち、上位の経路

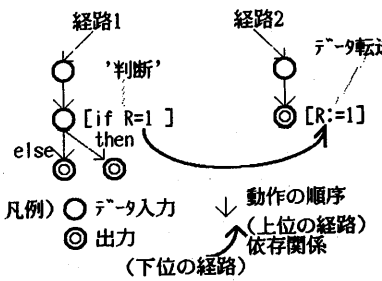
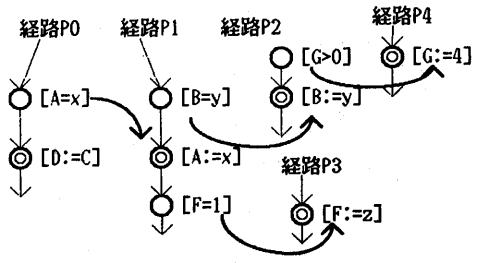


図6　経路の依存関係



注)A,…,G は資源 ,x,y,z は資源の値

図7　依存関係の連鎖

からテストデータを作っていき、テストデータ生成済みの経路を上位に持つ経路を次の生成対象とする。上位の経路のテストデータと該当経路のテストデータを連結して、依存関係を考慮したテストデータとする。これを最下位の経路まで繰り返す。

(イ)上記の手順では、図９のように依存関係が環状に閉じている場合は経路間の依存関係の上下が決められない。そこで、対象のハードウェアの動作によるデータ転送に代わり、外部から直接データ転送する資源を決め、これを見掛け上の上位の経路とし(ア)を行う。

４．３　入力データの組み合わせ

すべての設計誤りを検出できるテストデータを作るには、入力データのすべての組み合わせと、その印加順序のすべてを尽くさなければならない。しかし、大規模なハードウェアではテストデータの量は膨大となる可能性があるため、単一の設計誤りを検出することを目標とする。

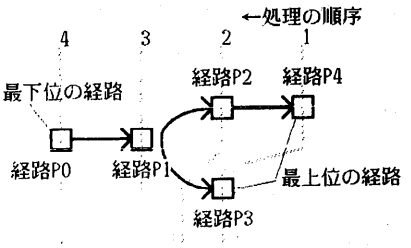


図８　依存関係の探索順序

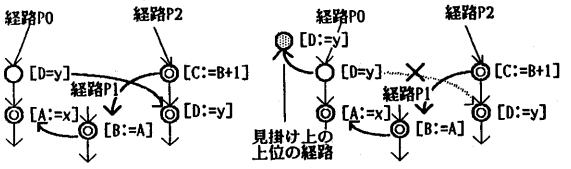


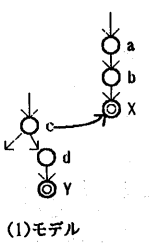
図９　閉じた依存関係

(1)規定された入力データに対して期待される出力を確認する("正常ｹｰｽ")外に、規定された以外の入力データによる出力("例外ｹｰｽ")を調べられるようにするが、それぞれ、以下のように選択する。

(ア)正常ケース：複数個の入力データとそれによって生じる出力とを考える。そこでは、すべての入力データが揃ったとき("論理積型入力")に出力が生じる場合と、いずれか１つの入力データでもあれば("論理和型入力")該当の出力が生じる場合がある。論理積型入力に対してはその入力データすべてと出力の組み合わせをテストデータとする。論理和型入力に対しては、個々の入力データと出力の組み合わせをテストデータに選ぶ。

(イ)例外ケース：資源の値の'判断'毎に、転送する値を規定と異なるものにして、次のデータ転送が行われないことを調べる入力データをテストデータとする。

(2)ループを含む経路については、ソフトウェアの機能テストデータにおいて提案されている[10]次の規則を採用する。すなわち、ループを通らない経路とループを１回経由する経路とに分けて、それぞれについてテストデータを作る。



No.	入力 a	入力 b	出力/次の入力 X/c	入力 d	出力 Y	備考
1	1	1	1	1	1	正常ｹｰｽ
2	1	0	0	1	0	例外ｹｰｽ
3	0	1	0	1	0	同上
4	0	0	0	1	0	×
5	1	1	1	0	0	例外ｹｰｽ
6	1	0	0	0	0	×
7	0	1	0	0	0	×
8	0	0	0	0	0	×

注)'1'：入出力がある
'0'：入力がないか、出力が生じない
×：この入出力の組み合わせは選択されない
(2)左記のモデルから作られるテストデータ

図１０　テスト入力の組み合わせ規則

5 考察

5．1 仕様記述言語

表1に示す2種類のプログラム制御装置の仕様を対象に記述実験を試み、以下を確認できた。

(1)記述性

属性定義は例外処理などの補足的機能を記述することを目的としているので、実用機のようなフルセットの機能を備えるハードウェアの記述に有効に使える。これは、アーキテクチャ確認用の実験機(装置B)では属性定義の比率が小さいが、フルセットの機能を網羅した装置Aでは、属性定義が一定の比を占めていることから推定できる。さらに、手続き型記述と宣言型記述が使われる割合は、類型的な動作の数に依存する。例えば命令ｺｰﾄﾞ数の比は装置Aが2.5に対し装置Bが1であり、それが記述量の比の違いとして表われている。

(2)記述時間

3．3で述べた専用エディタではコマンドの繰り返しや図面の部分流用等が可能で編集作業が容易にできるため、図面当たりの記述時間は約15分であり、それらの機能がない場合に比べ記述時間は、約65％に短縮できた。

5．2 テストデータ自動生成

(1)テストデータ量

本手法では、①全ての経路をカバーして因果関係を取り出すこと、②一定の規準に従って例外ケースのテストデータも生成することから、人手設計並のテストデータが生成できる。ここで、入力データの設定と出力の確認が連続して行えるものを1つのテストデータと定義すると、本手法によるテストデータ量は以下のように推定できる。

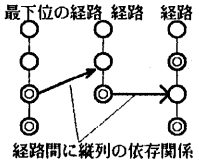
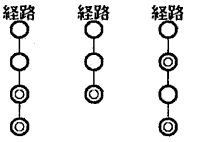
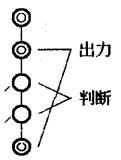
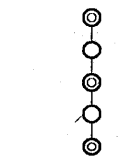
①正常ケースでは、1つの経路に対して1つのテストデータが生成される。ただし、依存関係を持つ経路がある場合、それらは連結されて1つのテストデータが生成される。したがって、依存関係を持たない経路数と最下位の経路数とによってこのテストデータ量が決まる。このテストデータ量が最少となるのは、すべての経路が縦列に依存関係を持ち、最下位の経路が1の場合であり、テストデータ量は1となる。一方、最多となるのはどの経路も依存関係を持たない場合であり、テストデータ量は経路数に等しい。動作の開始時点から調べて'判断'ごとにあらたに1つの経路が取り出されるので、機能仕様に含まれる'判断'の数をBとすると、経路数はB+1である。すなわち、最多のテストデータ量はB+1である。

②例外ケースでは、経路の中で2つの

表1 仕様記述言語の記述実験例

言語の構成 ＼ 実験対象			記述量の割合[%] 装置A	装置B
ｲﾝﾀﾌｪｰｽ資源			14	13
ﾌﾟﾛｾｽ	属性定義部		10	1
	機能定義部	手続き的記述	11	55
		宣言的記述	65	31

表2 テストデータ量の推定

	最少	最大
正常ｹｰｽ	最下位の経路 経路 経路  経路間に縦列の依存関係 ﾃｽﾄﾃﾞｰﾀ量＝1	経路 経路 経路  経路間に依存関係がない ﾃｽﾄﾃﾞｰﾀ量=B+1 B:判断の数
例外ｹｰｽ	 出力 判断 全ての判断が唯一の出力間に集中 ﾃｽﾄﾃﾞｰﾀ量=B+1	 1組の出力間の判断は1つ以下 ﾃｽﾄﾃﾞｰﾀ量<B(B+1)

'出力'間に'判断'が挟まった場合毎に1つのテストデータが生成される。したがって、テストデータ量が最少となるのは、すべての'判断'が唯一の'出力'同士間にある場合で、テストデータ量は経路数に等しい。一方、最多となるのは1組の'出力'間の'判断'の数が1つ以下の場合であり、テストデータ量は、各経路に属する'判断'の数の総和となる。1つの経路に属する'判断'の数はB以下であり、経路数は(B+1)なので、テストデータ量はB(B+1)以下である。

③以上からすべてのテストデータ量は、最少の場合にB+2となる。対象の機能仕様の中にn個の記述要素(ﾃﾞｰﾀ転送動作)があり、その中に一定の割合で'判断'が含まれると仮定すると、最少のテストデータ量はO(n)である。一方、最多の場合には$(B+1)^2$以下であり、同様に$O(n^2)$以下となる。

図11の試行例で、生成されるテストデータ量が$O(n)$と$O(n^2)$の範囲に収まっており、上記の解析が裏付けられている。

(2)生成時間

生成時間は、①経路を取り出す時間、②経路毎に因果関係を対応付ける時間、③依存関係を見つける時間、および④因果関係からテストデータを作る時間の総和である。それらは、経路間の依存関係がいくつ存在するか、'判断'が経路上にどのように分布しているか等によって決まる。それらの要因をすべて考慮して解析的に生成時間を推定するのが困難なため、ここでは前項と同一の記述例に対する実測値を求めた。その結果、図12に示すように$O(n^4)$の範囲の時間で生成できた。

(3)設計工数の削減効果

方式検討から論理設計までのハードウェア設計において、機能検証用のテストデータ作成を人手で行った例から、それに全工数の約1割を要すると推定される。製造検査用のテストプログラム設計においても、テストデータ作成を人手でおこなった例では、それに要する工数は全工数の約5割と考えられる。これらの工数は、本手法による自動化により大部分の削減が期待できる。

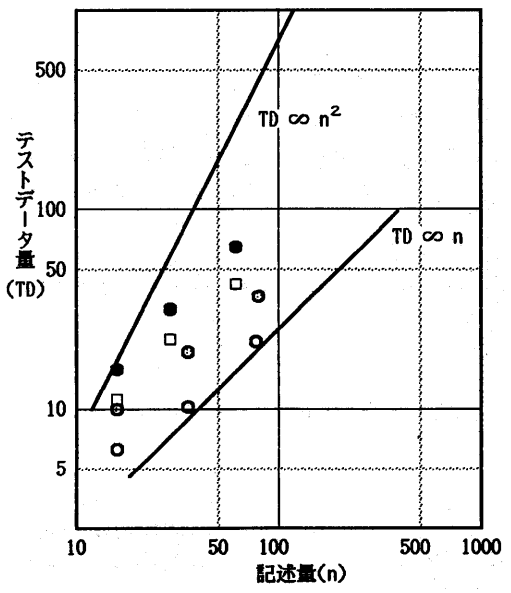


図11 テストデータ量

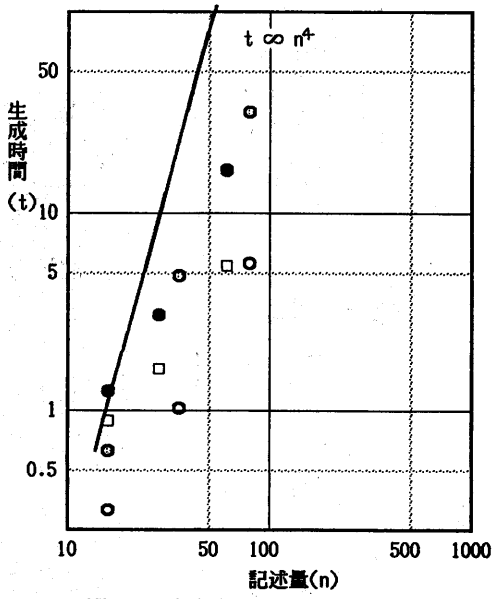


図12 生成時間

6 むすび

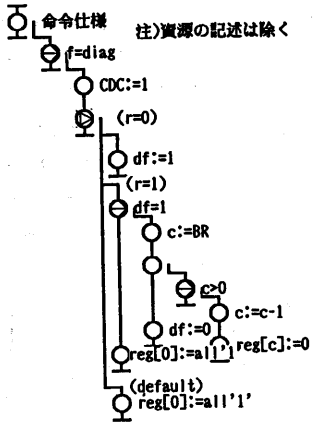
ハードウェアの機能仕様を記述する言語と、その記述からテストデータを自動生成する手法について報告した。

本手法によれば、プロセッサなどの機能仕様を効果的に記述でき、また、生成されるテストデータは人手設計に近く、その量は記述量の２乗以下となる。

現在、この仕様記述言語を入力として下位の機能記述に展開していく手法の検討を進めており[11]、それを本システムに組み込んで仕様記述段階からの設計支援システムとして統合することが今後の課題である。

最後に、本検討に当たり、御指導を賜わった、NTT電気通信研究所　基幹交換研究部　処理方式研究室　浜田泰昭室長、脇村慶明主幹研究員および室員各位に深謝いたします。

[参考文献]
(1)田代、津田:"ﾃｽﾄ用高級言語TPL"、信学論(D),J63-D,11,pp.923-930(昭55-11)
(2)菊地、村上、平川他:"LSIﾃｽﾄﾌﾟﾛｸﾞﾗﾑの自動発生"、第23回情処全大、1D-7(昭56後期)
(3)K.Lai:"Test Program Commpiler-A High Level Test Program";ICCAD'83 pp.30-31(1983)
(4)J.A.Darringer:"The Appricaton of Program Verification Techniques to Hardware Verification",16th DA Conf.,pp.375-381(1979)
(5)丸山文宏:"ﾊｰﾄﾞｳｪｱの機能設計段階における検証"、情処学論、21、5、pp.493-503(昭55-5)
(6)G.L.Smith,R.J.Bahnsen,H.Halliwell:"Boolean Comparison of Hardware and Flowcharts",IBM J.RES.DEVELOP.,26,1,pp.106-116(1982)
(7)小林、若林:"非手続き的記述をとりいれた図的ｱｰｷﾃｸﾁｬ記述言語";信学技報EC84-48(1984)
(8)若林、小林:"ｱｰｷﾃｸﾁｬ記述ｸﾞﾗﾌｨｯｸｴﾃﾞｨﾀ";第30回情処全大、2J-1(昭60前期)
(9)小林一夫:"ﾊｰﾄﾞｳｪｱﾃｽﾄ設計知識を用いたﾃｽﾄﾃﾞｰﾀ生成法";信学技報EC85-1(1985)
(10)古川、野木、徳永:"AGENT:機能ﾃｽﾄのためのﾃｽﾄ項目生成の一手法"、情処学論、25、5、(昭59-9)
(11)小林、若林:"設計ﾌﾟﾗﾝを用いたﾚｼﾞｽﾀﾄﾗﾝｽﾌｧ合成法";情処研報86-DA-33-2(1986)




```
if (f=diag)
  {
   CDC=1;
   switch (r) {
     case '0': df=1;
     case '1': if (df=1)
                 {
                  c=BR;
                  while (c>0) {
                    c=c-1;
                    reg[c]=0;
                  }
                  df=0;
                 }
                 else
                  reg[0]=all'1';
     default: reg[0]=all'1';
   }
  }
```

上図の説明

付図1　仕様記述例

付表1　適用結果

<No>	<test_phase>	<input>		<output>
			phase 1-1	
1	phase1-1-1	(f=diag)		(cdc:=1)
2	phase1-1-2	#((r=0))^#((r=1))		(reg[0]:=all_1)
			phase 1-2	
3	phase1-2-1	#((f=diag))		*
			phase 1-3	
4	phase1-3-1	(f=diag)		*
5	phase1-3-2	(r=0)		(df:=1)
6	phase1-3-3	(f=diag)		*
7	phase1-3-4	(r=1)^(df=1)		(c:=br)
8	phase1-3-5	#((c>0))		(df:=0)
			phase 1-4	
9	phase1-4-1	(f=diag)		*
10	phase1-4-2	(r=0)		*
11	phase1-4-3	(f=diag)		*
12	phase1-4-4	(r=1)^(df=1)		*
13	phase1-4-5	(c>0)		(c:=c-1)
14	phase1-4-6	*		(reg[c]:=0)
15	phase1-4-7	*		(df:=0)
			phase 1-5	
16	phase1-5-1	(f=diag)		*
17	phase1-5-2	(r=0)		*
18	phase1-5-3	(f=diag)		*
19	phase1-5-4	(r=1)^(df=1)		*
20	phase1-5-5	(c>0)		*
21	phase1-5-6	(f=diag)		*
22	phase1-5-7	(r=1)^#((df=1))		(reg[0]:=all_1)
			phase 1-6	
23	phase1-6-1	#(f=diag)		#(cdc:=1)
			phase 1-7	
24	phase1-7-1	(f=diag)		(cdc:=1)
25	phase1-7-2	#(#(r=0))v#(#(r=1))		#(reg[0]:=all_1)