高水準設計検証の一方式†

高原 厚†† 南谷 崇††

LSI/VLSI による実現を前提としたディジタルシステムの設計に誤りが存在する場合，それらはできるだけ早期に発見されることが望ましい．これまでに提案されている設計検証方式の多くは，レジスタ転送レベル以下の論理設計を対象としている．抽象的な仕様から具体的な実現へ至るトップ・ダウン方式の設計過程を考えた場合，できるだけ早期に設計誤りを発見するという立場からは，より抽象度の高い設計表現のレベルにおいてもその設計の正しさを検証することが望ましい．本論文では，アルゴリズムレベルの表現を仕様記述とし，機能ブロックを用いたハードウェア記述を実現記述とした高水準設計検証を，補仕様という新たな概念を用いて定式化し，それに基づいた検証方式を提案する．これは，実現記述が補仕様記述を満たしていることを検証するものである．本検証方式では，ハードウェア・モデルとして Milner の CCS を用いる．このモデル上では実際の設計誤りは三つの設計誤りモデルとして表現される．検証は，補仕様記述と実現記述をハードウェア・モデル上で合成したものが設計誤りモデルにあてはまること無く動作することを示すことにより行われる．この方式では，簡潔なアルゴリズムで設計検証を行うことができる．

1. まえがき

LSI/VLSI による実現を前提としたディジタルシステムの設計に誤りが存在する場合，それらはできるだけ早い段階で発見されることが望ましい．論理設計の誤りを検出する方法として論理シミュレーションが現在広く用いられている．これは，あり得るすべての入力組み合わせに対してシミュレーションを実行しない限り，論理設計の正しさを保証することはできない．このため，論理設計の正しさを形式的に検証するための提案がいくつかなされている．それらは，

- ソフトウェアの検証方式を利用したもの[1]
- ハードウェアの基本要素を公理として示し，それらによって構成されるシステムが正しいことをその公理を用いて検証するもの[2)~4)]
- テンポラル・ロジックなどある論理体系または，ハードウェア・モデルを用いるもの[5)~7)]

等に分けられる．これらの検証方式は，レジスタ転送レベル以下の論理設計を対象にしたものである．抽象的な仕様から具体的な実現へ至るトップ・ダウン方式の設計過程を考えた場合，できるだけ早期に設計誤りを発見するという立場からは，より抽象度の高い設計表現のレベルにおいてもその設計の正しさを検証することが望ましい．本論文では，まず高水準設計検証の定式化を行い，次に Milner の CCS[8] により記述された補仕様記述，対応記述，実現記述を用いた検証方式について述べる．

2. 高水準設計検証の定式化

2.1 高水準設計検証の目的

ディジタルシステムの設計とは，所望の動作仕様を記述した仕様記述から，一定の設計制約の下で与えられた基本素子で実現される物理的な構造を記述した実現記述への変換と考えることができる．設計検証は，この仕様記述から実現記述への変換が正しく行われていることを示すことである．本論文では，アルゴリズムレベルの表現を仕様記述とし，機能ブロックを用いたハードウェア記述を実現記述とした，高水準設計検証を考える．アルゴリズムレベルの仕様記述においては，ハードウェアとしての実現方法を意識せずに所望の動作仕様が記述される．これに対し，実現記述ではデータパスと制御が明確に区分されたハードウェアの実現方法が示される．高水準設計検証の目的は，実現方法を意識せずに記述された動作仕様が，データパス，制御に分けて構成されたハードウェア上で，正しく実現されていることを示すことである．

2.2 対応記述

仕様記述と実現記述のみでは，以下に述べるように設計検証を行う情報としては不十分である．

いま，次のように仕様記述が与えられる回路を設計することを考える．

$$\text{IF } C \text{ THEN } A \Rightarrow D \text{ ELSE } B \Rightarrow D \qquad (2\text{-}1)$$

これは，C で表された条件によって，A, B のどちらかのデータを D に転送するというものである．いま，

† An Approach to Higher Level Design Verification by ATSUSHI TAKAHARA and TAKASHI NANYA (Department of Computer Science, Faculty of Engineering, Tokyo Institute of Technology).

†† 東京工業大学工学部情報工学科

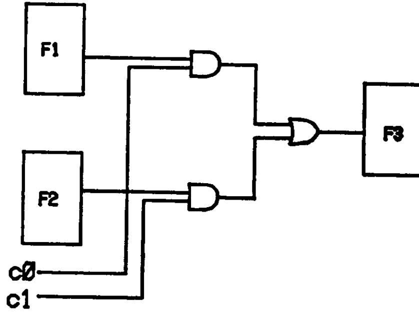


図 1 式(2-1)に対する実現
Fig. 1 An implementation of a circuit conforming to its Specification (2-1).

表 1 仕様記述と実現記述の関係
Table 1 Relation between specification description and implementation description.

対応 1		対応 2	
仕様記述	実現記述	仕様記述	実現記述
A	F1	*A*	F1
B	F2	*B*	F2
C	F3	*C*	F3
C=TRUE	C0=1, C1=0	*C*=TRUE	C0=0, C1=1
C=FALSE	C0=0, C1=1	*C*=FALSE	C0=1, C1=0

実現記述として**図1**の回路が与えられたとする．図1に示されていることは，

① C0=1 ならば，F1 から F3 へデータを転送

② C1=1 ならば，F2 から F3 へデータを転送

ということである．これだけでは，式 (2-1) で示される，*A*, *B*, *C*, *D* が実現記述のどの構成要素に対応しているかということが示されないので，図1の実現記述は式 (2-1) の仕様記述を満たしているかどうかを判断することができない．なぜなら，**表1**に示される二つの対応関係を考えた場合，対応1では「仕様を満たしている」，対応2では「仕様を満たしていない」という二つの異なる結論が得られるからである．

この例のように，実現記述が仕様記述を満たしているかどうかということを示すためには，この二つの記述間の対応関係を表す写像（これを対応記述と呼ぶ）を与えることが必要である．

従来のレジスタ転送レベル以下を対象とする論理設計検証のほとんどは，すでにデータパス，制御の分離が行われた仕様記述と，上位の仕様記述の各ブロックに対応して詳細化された実現記述を対象とした検証である．このため，仕様記述，実現記述間の対応関係を与えることが容易である．しかし，高水準設計検証においてはこのような対応記述が必ずしも明確にならない場合が考えられる．これは，実現記述においては，データパス，制御が明確に区分されているが，仕様記述ではこれらの区分が明確ではないことによる．また，対応記述を考えると，仕様記述における各変数 (*A*, *B* 等) について一つ一つ対応関係を示す必要がある．VLSI により実現される大規模なシステムの検証を考える場合，対応記述で示される対応関係は膨大な数となりうる．そのため，対応関係の数を少なくすることが望まれる．

2.3 仕様と補仕様

このような対応記述における問題点を考えると，高水準設計検証では従来からの仕様記述の方法をそのまま用いることは難しい．そのため，仕様記述，実現記述間の写像が必ず決まり，また，対応関係の数を減らすような仕様記述を考えることが必要である．そこで，この“仕様”について考えてみると，“仕様”として記述する立場には二つの立場がある．一つは，実現すべきシステムの動作を，制御の流れ，データ形式と位置関係，演算の種類等システムの内部動作に着目して記述する立場である．もう一つは，システムを外部から見た時満たされるべき入出力条件を記述する立場である．

ここで，OCCAM[9] を用いてスタックの仕様をこの両者の立場から記述した例を考えてみよう．まず，図2(1)では，前者の立場からスタックの仕様が以下のように記述されている．

① ポート ctrl に与えられたデータ c の値が pop であれば stack の pointer で示される値をポート out に出力し，pointer をデクリメントする．

② ポート ctrl に与えられたデータが push の場合には pointer をインクリメントした後，ポート in に与えられたデータを stack に格納する．

これに対し，後者の立場から仕様を記述したものが図2(2)である．ここでは，スタックに外界がどのように作用するかを示している．変数 stack は最初に x1〜x*n* の値を持つように初期化されている．そして，以下の動作のどちらかを非決定的に行うことが許されるシステムとしてスタックの仕様が表現されている．

① ポート ctrl に pop を出力し，out に受け取ったデータが stack の pointer の指す値と等しいことを確認する．この時，等しくなければそれは仕様からはずれた動作をしたことを意味する．

② ポート ctrl に push という値を出力し，さらに

```
PROC stack_spec (CHAN in,out,ctrl) =
  VAR pointer,stack[]
  SEQ
    pointer := 0
    WHILE TRUE
      VAR c,data
      SEQ
        ctrl?c
        IF
          c = pop
            SEQ
              out!stack[pointer]
              pointer := pointer - 1
          TRUE
            SEQ
              in?data
              pointer := pointer + 1
              stack[pointer] := data
```

（1） 仕様記述
(1) Specification description.

```
PROC stack_cospec (CHAN in,out,ctrl) =
  VAR pointer,stack[],data
  SEQ
    stack[] = {x1,x2,...,xn} /* initialization of stack */
    pointer := 0
    WHILE TRUE
      ALT
        SKIP
          SEQ
            ctrl!pop
            out?data
            pointer := pointer - 1
            IF
              data = stack[pointer]
              TRUE
                /* error */
        SKIP
          SEQ
            ctrl!push
            in!stack[pointer]
            pointer := pointer + 1
```

（2） 補仕様記述
(2) Co-specification description.

図 2 仕様
Fig. 2 Specification.

ポート in に pointer の指す stack の値を出力する．

このように，スタックに push, pop の命令を与えた時にどのような値がスタックから出力されるべきかを示すことにより仕様を表現している．

これまで仕様と呼んでいたものは，このような二つの立場を含んでいると考えられる．本論文では，この立場を次のように区別する．

仕様記述：ハードウェアが内部でどのように動作するべきかという立場で設計目標を与える

補仕様記述：実現されるハードウェアが外部からどのように見えるべきかという立場で設計目標を与える

この二つの記述の違いは以下のように考えられる．ある実現されたハードウェアが与えられた場合，それに対応する環境が存在する．一方，実現されたハードウェアより抽象度の高いレベル（仕様のレベル）においても，実現されたハードウェアに対応する部分と，環境に対応する部分を考えることができる．ここでは，抽象度の高いレベルで実現されたハードウェアに対応する部分を仕様記述，環境に対応する部分を補仕様記述としている（図3）．先の例では，図2(1)が仕様記述，図2(2)が補仕様記述である．

このように，補仕様記述はハードウェアのおかれる環境の抽象化表現である．補仕様記述では，アルゴリズムレベルの仕様をハードウェアの入出力関係により表現する．この時，対応記述としてはハードウェアの外部に対する入出力について，それが補仕様記述における入出力とどのように対応するかを示すのみでよい．そのため，入出力に関してのみ対応関係を示すことにより仕様記述から実現記述への写像が与えやすい．また，対応記述は入出力に関して示せばよいので対応関係の数も少なくなる．

2.4 モデル化

前節までに，設計検証には，補仕様記述，対応記

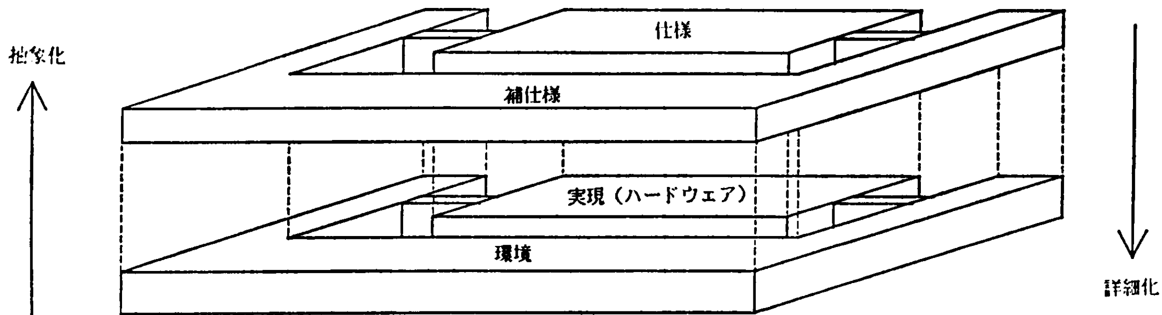


図 3 仕様-補仕様
Fig. 3 Specification and Co-specification.

述，実現記述が必要であることを述べた．しかし，これらの記述を検証に直接用いることは困難である．補仕様記述と実現記述では，抽象度が異なっているからである[10]．設計検証を行うためにはこの両者を同じレベルにおいて議論することが必要である．そこで，設計検証を目的とした一つのハードウェア・モデルを考え，補仕様記述，対応記述，実現記述をそのモデルに変換する．そして，設計検証とはそのモデル上において実現記述が補仕様記述を満たしていることを示すことである．

一方，各記述をハードウェア・モデルに変換するとともに，ハードウェア・モデル上における設計誤りを規定することが必要である．設計誤りには，制御，構造，インタフェース，タイミング等におけるものが考えられる．このような具体的な設計誤りを考えるのではなく，モデル化されたレベルにおける設計誤りを規定する．そして，実現記述が補仕様記述を満たしていることを，規定された設計誤りが存在しないことを示すことにより検証する．

さらに，ハードウェア・モデルを考える場合にはそのモデルの解析が容易であることが望まれる．設計検証においては対象となるハードウェアが到達する状態をすべて知ることが必要である．しかし，ハードウェアの事象，時間ごとの状態をすべて考えることは困難である．そこで，ハードウェア・モデルに求められることは，対象とするハードウェアの動作を損なわず，最大限の抽象化が行えることである．

2.5 高水準設計検証

これまでに述べてきた高水準設計検証は以下のようにまとめられる．

【高水準設計検証】

① 入力として補仕様記述，対応記述，実現記述を用いる．

② これらの記述をあるハードウェア・モデルに変換する．

③ そのモデル上で定められた設計誤りが存在しないかどうかを調べる（図4）．

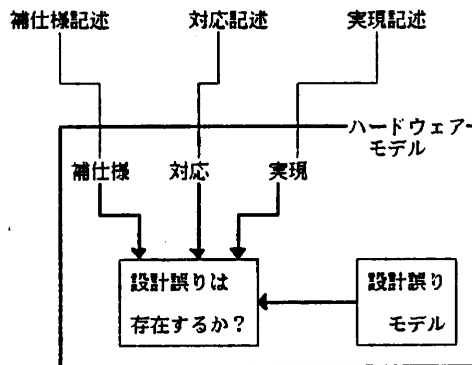


図 4 形式的論理設計検証
Fig. 4 Formal design verification.

3. CCS を用いた設計検証

本論文では，ハードウェア・モデルとして Milner により提案された CCS を用いた設計検証方式を提案する．ここでは，2.5 節で示した③にあたる，ハードウェア・モデル上で設計検証をどのように行うかについて説明する．

3.1 CCS の概要

ハードウェア・モデルとして解釈した場合の CCS の概要とその利点を述べる．

CCS ではシステムの動作をシステムと外界とを結ぶポートの動作として表現する．入力ポート α に対し，関数 f を作用させ，出力ポート β にその結果を与えるモジュール A は次のように表現される．

$$A=\alpha\ x.\ \bar{\beta}\ f(x).A \qquad (3\text{-}1)$$

CCS ではポートは α または，$\bar{\alpha}$ という形式で表現される．α は入力ポートを $\bar{\alpha}$ は出力ポートであることを示している．次に，ポート α, β に与えられた値をポート γ に出力するスイッチ S を考えてみる．

$$S=\alpha\ x.\bar{\gamma}\ x.S+\beta\ x.\bar{\gamma}\ x.S \qquad (3\text{-}2)$$

ここで，＋操作は S のモジュールでは二つの動作がコンカレントに可能であることを示している．S では，α または β に与えられた値を $\bar{\gamma}$ に出力する．

次に，二つのモジュール A と S を組み合わせて一つのシステム SY を構成する（図5）．A と同じ機能を持つモジュール $A1$, $A2$ はポートの名前を変更する RELABELING 操作により式 (3-1) から生成することができる．

$$A1=A[\alpha 1/\alpha, \beta 1/\beta]$$
$$=\alpha 1\ x.\overline{\beta 1}\ f(x).A1 \qquad (3\text{-}3)$$
$$A2=A[\alpha 2/\alpha, \beta 2/\beta] \qquad (3\text{-}4)$$

また，このように生成されたモジュール $A1$, $A2$

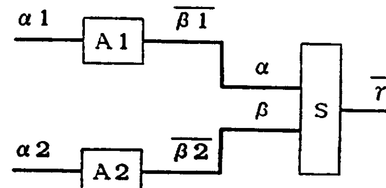


図 5 ハードウェア構成
Fig. 5 Hardware configuration.

と式(3-2)の S を合成するCOMPOSITION操作により，図5に示されるようなシステム SY が得られる．

$$SY=(A1|A2|S[\beta1/\alpha, \beta2/\beta]) \quad (3\text{-}5)$$

ここで，S に対してRELABELING操作が行われている．CCSではポート間の通信は名前が同じである対応するポート間において行われる．また，ポートが通信可能なのは各表現の先頭に示されたポートである．式(3-3)では $\alpha1$ が通信可能である．Sの入力ポート α, β を $\beta1, \beta2$ に変えることにより，$A1, A2$ の出力がSの入力ポートに与えられることを示すことができる．このように，CCSではシステム間の接続関係をポートの名前のみで表現することができる．そして，ポートにより接続関係が示されるため，モジュールの合成を式(3-5)のようにCOMPOSITION操作により簡潔に表現することができる．

次に，このCOMPOSITION操作により生成された SY を解析することを考える．この時，$\beta1, \beta2$ は SY の内部的なポートであるから，SY を外界から見た場合には $\beta1, \beta2$ というポートの動作は隠されることが望ましい．このように内部的なポートの動作を隠す操作は，RESTRICTION操作と呼ばれる．SY に対し，$\beta1$ と $\beta2$ の動作を隠すRESTRICTION操作は式(3-6)のように表現される．

$$SY=(A1|A2|S[\beta1/\alpha, \beta2/\beta])\backslash\beta1\backslash\beta2 \quad (3\text{-}6)$$

これにより，外界に対するシステム SY の動作のみ($\alpha1, \alpha2, \gamma$ の動作)を示すことができる．また，動作が継続しないことはNILで表現される．

CCSではこのように合成されたシステムの動作を解析する場合に以下の展開定理(expansion theorem)が用いられる．

【展開定理】

$B=(B_1|\cdots|B_m)\backslash A$ とする．

$$B=\sum\{g.((B_1|\cdots|B_i'|\cdots|B_m)\backslash A);\quad (3\text{-}7)$$

ここで，$B_i=g.B_i'$, $g\notin A\}$

$$+\sum\{\tau.((B_1|\cdots|B_i'|\cdots|B_j'|\cdots|B_m)\backslash A);$$

ここで，$B_i=\bar{\alpha}.B_i'$, $B_j=\alpha.B_j'$, (3-8)

$i\neq j$, $\alpha\in A\}$

【第1ステップ】

$\alpha1 \quad x.(\overline{\beta1} \quad f(x).A1|A2|S)\backslash\beta1\backslash\beta2$

$+$

$\alpha2 \quad x.(A1|\overline{\beta2} \quad f(x).A2|S)\backslash\beta1\backslash\beta2$

【第2ステップ】

$\alpha1 \quad x.$

$\quad(\alpha2 \quad x.(\overline{\beta1} \quad f(x).A1|\overline{\beta2} \quad f(x).A2|S)\backslash\beta1\backslash\beta2$

$\quad +$

$\quad \tau.(A1|A2|\overline{\gamma} \quad f(x).S)\backslash\beta1\backslash\beta2)$

$+$

$\alpha2 \quad x. \cdots$

【第3ステップ】

$\alpha1 \quad x.(\alpha2 \quad x. \cdots$

$\qquad +$

$\qquad \tau.(\overline{\gamma} \quad f(x).(A1|A2|S)\backslash\beta1\backslash\beta2$

$\qquad\quad +$

$\qquad\quad \alpha1 \quad x.(\overline{\beta1} \quad f(x).A1|A2|S)\backslash\beta1\backslash\beta2$

$\qquad\quad + \quad \cdots)$

$\quad)$

$+$

$\alpha2 \quad x. \cdots$

図6 展開定理による SY の解析
Fig. 6 The analysis of SY with expansion theorem.

この展開定理を適用すると，システムのポートの動作順序を導出することができる．この定理を，先の式(3-6)に順次適用した結果を図6に示す．まず，第1ステップでは展開定理の式(3-7)が適用され，RESTRICTIONされていないポート $\alpha1, \alpha2$ のそれぞれが通信した結果が現れる．このため第1ステップでは二つの場合が導出されている．次に，第2ステップではRESTRICTIONされているポート $\beta1$ の動作が行われる．展開定理の式(3-8)が適用され，ポート $\beta1$ 間で通信が行われ，この結果として τ が現れる．τ はシステム内部のポート間で通信が行われたことを示すものである．そのためシステムの外部から観測した場合には，どのポートが通信したかということは問題ではないので τ により通信が行われたことのみが示される．また，τ が任意の個数連続することを τ^{*} と表現する．本論文では，COMPOSITIONにより表現されるものをシステムの状態と呼ぶことにする．このようにして展開定理を順次適用することにより，システムの動作を解析することができる．以上の特徴をもつCCSを用いてどのような方式で検証を行うかを次節で示す．

3.2 CCSモデルによる形式的設計検証

本論文の提案する設計検証方式の概念を2.3節で述べたスタックを例にして説明する．補仕様も，CCSにおいてスタックとしてモデル化できる．この補仕様を

表現するスタックは，次の点が一般のスタックと異なる．

① 補仕様を表現するスタックの初期状態は，スタックが満杯の状態である．

② PUSH 動作，POP 動作を非決定的に繰り返し行う．

③ スタックにデータを積む場合にはそのデータが初期状態での補仕様のスタックの状態を保存するようなデータのみを積む．それ以外のデータが与えられた場合には補仕様と実現間に矛盾があることとなり，動作を停止する．

次に，補仕様の入力を実現されたスタックの出力に，補仕様の出力を実現されたスタックの入力に接続する．このようにして二つのスタックを合成して一つのシステムを構成する．ここで，設計誤りとしてスタックに与えられたデータが喪失する場合を考え，これがどのように発見されるかを示す（図7）．このような場合，まず補仕様を表すスタックから $D1$ を PUSH すると，実現されたスタックではそのデータが失われてしまうのでスタックの状態は変わらない．次に，POP 動作を行うと，補仕様側のスタックには $D1$ という値が入力されることが要請される．ところが，$D1$ は失われているため $D0$ という値が補仕様側に入力される．そこで，$D1 \neq D0$ となり補仕様側のスタックにおける③の動作により，システムの停止として先の設計誤りを発見することができる．

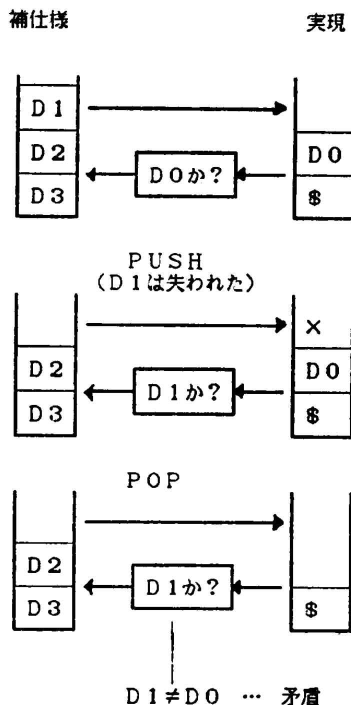


図 7　スタックの検証
Fig. 7　Verification of stack.

このように，本設計検証方式は実現記述と補仕様記述を合成してひとつのシステムとして動作させた時に，その動作が矛盾無く継続することを示すことにより，実現記述が補仕様記述を満たしていることを示すものである．補仕様と実現されたハードウェアとの合成を，CCS として表現すると以下のようになる．

CO-SPEC : CCS による補仕様記述
IMPLEMENT : CCS による実現記述
RELATION : CCS による対応記述
CO-SPEC の入力ポート : CS-Ii $(0 \leq i \leq N)$
CO-SPEC の出力ポート : $\overline{\text{CS-O}i}$ $(0 \leq i \leq M)$
IMPLEMENT の入力ポート : IMP-Ii $(0 \leq i \leq N)$
IMPLEMENT の出力ポート : $\overline{\text{IMP-O}i}$ $(0 \leq i \leq M)$
RELATION の入力ポート : IMP-Oi ∪ CS-Oi
RELATION の出力ポート : $\overline{\text{IMP-I}i}$ ∪ $\overline{\text{CS-I}i}$

$$S = (\text{CO-SPEC} \mid \text{RELATION} \mid \text{IMPLEMENT}) \backslash A \quad (3\text{-}9)$$

$$A = \{\text{IMP-I}i\} \cup \{\text{IMP-O}i\} \cup \{\text{CS-I}i\} \cup \{\text{CS-O}i\}$$

3.3　設計誤りのモデル化

本検証方式では，式 (3-9) の S の動作が継続できない場合には実現記述が補仕様記述を満足しないことを意味する．これには以下の三つの場合がある．

1)　デッド・ロックとなる場合

デッド・ロックとなるのは，式(3-9) の S で RESTRICTION されているポートにおいて，入力ポートと出力ポートが同時に通信可能となるものがない場合である．CCS ではポートの通信動作によってハードウェアの動作が表現されている．そのため，ハードウェアの動作の矛盾はポート間における通信の矛盾としてモデル化できる．

2)　COMPOSITION におけるある項が NIL となる場合

本検証方式では，実現されたハードウェアから与えられたデータが補仕様からはずれた場合設計誤りとなる．このことを CCS では，補仕様が NIL となることにより表現する．例えば，2.3 節で示される補仕様のスタックの入力が補仕様の変数 stack の次の値と等しくない場合には NIL となる．このように補仕様の入出力条件を満たさない場合，式(3-9)の S における ある項が NIL という状態となり設計誤りを示す．

3) τ 以外のものが導出された場合

RESTRICTION されていないポートが展開定理の適用の結果として現れることがある．これは補仕様としては全く考えていないポートであるためRESTRICTION には含まれず，展開定理の適用の結果として現れるものである．このように，式(3-9)の S に展開定理を適用した結果として τ 以外のものが現れる場合も設計誤りとなる．

以上三つの場合がハードウェア・モデルとして CCS を用いた設計検証における設計誤りモデルである．式(3-9)の S が上記の三つの場合になることなく動作する場合は設計誤りが存在しないことを示す．この場合，CCS では合成されたシステムから τ のみが出力されることになる．これより，CCS を用いた設計検証は次のように表すことができる．

CCS を用いた設計検証

$S=\tau^{\omega}$…補仕様を満たしている

$S\neq\tau^{\omega}$…補仕様を満たしていない

式(3-9)の S が τ^{ω} と等価であるかどうかは図8に示される検証アルゴリズムに従って展開定理を適用することにより示される．このアルゴリズムは展開定理のみを用いる簡潔なものである．

4. 検 証 例

3章で述べた検証アルゴリズムに基づき，設計検証システムを試作した．入出力部分は PASCAL（約 2,000 行）で，展開定理を適用する部分は LISP（約 1,000 行）を用いて記述されている．以下にこの検証システムを用いてシストリック・カウンタ[11]を検証した例を示す．

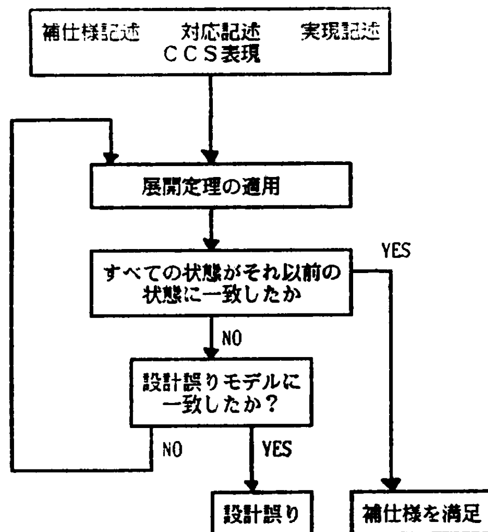
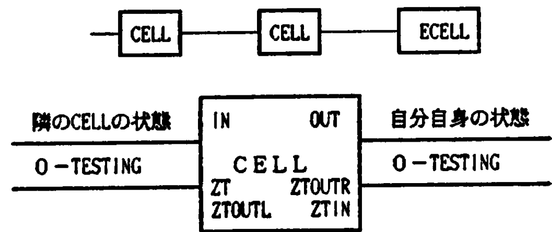


図 8 検証アルゴリズム
Fig. 8 Verification algorithm.

シストリック・カウンタ



各CELLのとりうる状態は以下のものである．

1) 0 ・・・ そのCELLの表す桁が0である
2) 1 ・・・ そのCELLの表す桁が1である
3) C ・・・ そのCELLの表す桁が右隣のCELLに対して桁上げをしたい場合
4) B ・・・ そのCELLの表す桁が右隣のCELLの表す桁から繰り下げを要求している場合

シストリック・カウンタの状態遷移

左のセル ＼ 自分の状態	B	0	1	C
B	0	B	0	B
0	1	0	1	0
1	1	0	1	0
C	C	1	C	1

図 9 シストリック・カウンタ
Fig. 9 Systolic counter.

4.1 シストリック・カウンタの CCS による記述

ここで検証するシストリック・カウンタはアップ・ダウン・カウンタである．このカウンタに対する命令は①カウント・アップ，②カウント・ダウン，③ゼロ・テストの三つである．

シストリック・カウンタは同様な CELL を直線状に配列して構成される．各 CELL の動作は，自分自身の状態と両隣の CELL の状態に依存する（図9）．CCS による実現記述は CELL の状態が0の場合は図10(1)となる．カウント・アップ，カウント・ダウンの場合には各 CELL は自分の状態をポート OUT に出力した後，入力ポート IN に与えられた値 X に応じて次の状態（CELL 0〜CELL 3）に遷移する．ZT は，ゼロ・テスト命令を受け取るポートである．この時，各 CELL は，その命令をまず右隣の CELL にポート ZTOUTR を用いて伝える．自分自身の右隣の CELL が0かどうかをポート ZTIN の値 X に受け取る．X

【実現記述】

```
CELL0 = - IN X.(IF X = C THEN *OUT C0.CELL1
                ELSE IF X = B THEN *OUT C0.CELLB
                ELSE *OUT C0.CELL0)
        + -ZT.*ZTOUTR.-ZTIN X.IF X = 0
                THEN *ZTOUTL 0.CELL0
                ELSE *ZTOUTL 1.CELL0;
        ⋮
```

（1）シストリック・カウンタに対する実現記述
(1) Implementation description for systolic counter.

【補仕様記述】

```
COSPEC = *INPUT UP.COSPEC1
       + *ZEROTST.-ZERORES X.IF X = 0 THEN COSPEC ELSE NIL;
COSPEC1 = *INPUT UP.COSPEC2 + *INPUT DOWN.COSPEC
       + *ZEROTST.-ZERORES X.IF X = 0 THEN NIL ELSE COSPEC1;
COSPEC2 = *INPUT DOWN.COSPEC1
       + *ZEROTST.-ZERORES X.IF X = 0 THEN NIL ELSE COSPEC2;
```

（2）シストリック・カウンタに対する補仕様記述
(2) Co-specification description for systolic counter.

【対応記述】

```
RELATION = -INPUT X.(IF X = UP THEN *OUTPUT C ELSE IF X = DOWN THEN
       *OUTPUT B ELSE *ZT.-ZTOUTL X.*ZERORES X).RELATION
```

（3）シストリック・カウンタに対する対応記述
(3) Relation description for systolic counter.

図 10　シストリック・カウンタの CCS による記述
本検証システムでは，A を $-A$，$\bar{A}$ を $*A$ と表記する．
Fig. 10 Description of systolic counter in CCS.

が0であれば自分自身も0であるので，0をポート ZTOUTL に出力する．そうでない場合には1を出力する．

シストリック・カウンタに対する補仕様記述は図10(2)となる．ここでは，2までのカウント・アップ，カウント・ダウンを行う．COSPEC, COSPEC 1, COSPEC 2 はカウンタ値が 0, 1, 2 の場合に対応する．ZEROTST はカウンタ値が0かどうかを確かめる命令であり，ZERORES はその結果を受け取るポートである．そして，その結果が0となるのは COSPEC の状態だけなので，それ以外の状態で0かどうかのテストが0になる場合は NIL という状態になる．

また，対応記述は図10(3)となる．補仕様記述における INPUT に与えられる UP, DU, ZTST を実現記述で扱う値 B, C, ZT に変換する．このようにして，対応記述では，補仕様記述と実現記述間の対応関係を示す．

4.2 検　　証

図 10(1)～10(3) に示される，実現記述，補仕様記述，対応記述から式(3-9)の S は式(4-1)となる．

```
(COSPEC|
  RELATION [OUTPUT/IN]|
CELL 0 [OUT/OUT 1,
  ZTOUTR/ZT 1, ZTIN/ZTOUTL 1]|
CELL 0 [IN/OUT 1, OUT/OUT 2,
  ZT/ZT 1, ZTOUTL/Z TOUTL 1,
  ZTOUTR/ZT 2, ZTIN/ZTOUTL 2]|
ECELL 0 [IN/OUT 2, ZT/ZT 2,
  ZTOUTL/ZTOUTL 2])
INPUT, OUTPUT, IN OUT 1,
  OUT 2, OUT 3, ZT 1, ZT 2,
  ZTOUTL 1, ZTOUTL 2,
  ZERORES, ZT, ZTOUTL,
  ZTOUTR 1                    (4-1)*
```

この式に展開定理を繰り返し適用することにより τ のみが導出され，第19 STEP ですべての状態がそれ以前の状態に一致する．この結果，式(4-1)が τ^{ω} と等価となり設計誤りの無いことが示される．

次に設計誤りが存在する場合，3.3 節で示した設計誤りモデルのどの場合として設計誤りが発見されるかを二つの例を用いて説明する．

第1の例として，設計誤りとして最右端の CELL について他の CELL と区別せずに同じ CELL を用いた場合について考える（式(4-2)）．式(4-1)では，最右端の CELL に ECELL 0 を用いたのに対し，ここでは他の CELL と同じように CELL 0 を用いている．

```
(COSPEC|RELATION [OUTPUT/IN]|
        ⋮
CELL 0 [IN/OUT 2, ZT/ZT 2, ZTOUTL/
  ZTOUTL 2])                  (4-2)
        ⋮
```

この場合，最右端の CELL 0 が動作した時点で，設計者が意図しなかったポートの動作が現れることによ

* 本検証システムでは，A を B に変える RELABELING を，$[A/B]$，また，RESTRICTION を $(\cdots|\cdots|\cdots)\underline{A, B, C, \cdots}$ と表記する．

り CELL 0 を用いた設計誤りを発見することができる.

第2の例として，設計誤りのなかで制御に矛盾がある場合を示す．本来は CELL 0 において C, B 以外の値が入力された場合は，次に CELL 0 に遷移しなければならないが，式(4-3)では CELLB に遷移してしまう.

```
CELL 0 = −IN X. (IF X=C THEN
            *OUT CO. CELL 1
            ELSE IF X=B THEN
            *OUT CO. CELLB
            ELSE *OUT CO. CELLB)    (4-3)
            +         ⋮
```

この場合は，補仕様記述からゼロ・テスト命令を入力すると，NIL という状態に到ることによりその矛盾を発見することができる.

4.3 他の検証方式との比較

CCS に基づいた設計検証方式はこれまでにも，Gordon[12]，Milne[13] らにより提案されているが，両者ともレジスタ転送レベル以下の設計を対象としたものである．Gordon の検証方式は，CCS 表現で記述された仕様と実現が等価であることを示すものである．しかし，対応記述を用いていないことから，仕様記述はすでに実現方法を明示していなければならない．すなわち，実現記述は仕様記述を同じレベルで書き変えたものである．このような方法を本論文で対象とする高水準設計検証に用いることはできない．なぜなら，高水準設計検証では，普通仕様記述において実現方法が明示されないからである．一方，Milne の方法は本検証方式と同様に仕様としてシステムに対する許される入出力動作を示す．そして，この仕様記述と実現記述を合成し，実現記述のみに現れるポートの動作を RESTRICTION したものの動作が仕様記述と等価であることを示すことにより検証を行うものである．しかし，これもやはり仕様記述により示される入出力表現が実現記述と同レベルであり，本検証方式のように仕様記述が実現記述とレベル的に異なるものではない．またこの方法では，等価性を示す場合，展開定理を用いて仕様記述と実現記述を合成したものを解析した後，さらに仕様記述と比較しなければならない．これに対し，本検証方式では補仕様において仕様から外れた動作が行われたかどうかの判断を含んでいるため，展開定理のみを適用する簡潔なアルゴリズムで設計検証を行うことができる.

しかし，我々の試作した実験システムでは先のカウンタの例においてセル数が4の場合，7 M の実行領域を必要とする（M-280 で UTILISP 使用時）．本稿で示したアルゴリズムは簡潔であるが，大規模なシステムの場合かなりの状態数となり効率よく τ^{ω} と等価であることを導出する方法が必要である．この問題点に対し，Milne は仕様としてある限られた入出力だけを与えることにより部分的な動作の検証を行うことを提案している．検証対象を制限することにより大規模なシステムに対する設計検証を行うことを容易にすることができる．このような点を今後検討することが必要である.

5. むすび

本論文では，実現方法を意識せずにアルゴリズムレベルで表現された仕様記述とデータパス，制御が明確に区分された実現記述を対象とした高水準設計検証を定式化し，それに基づいて CCS を用いた検証方式を提案した．補仕様記述と実現記述を合成しお互いの動作が矛盾無く続くことを調べることにより，実現記述が補仕様記述を満たしていることを示すものである．この検証方式の利点は，検証を CCS というハードウェア・モデル上で行うことにより，実際の設計誤りをいくつかの設計誤りモデルに帰着し，展開定理のみを用いる単純なアルゴリズムで検証を行うことができることである.

しかし，補仕様がどのようなハードウェアに対しても有効な表現方法であるかの検討，また，検証アルゴリズムにおいて τ^{ω} と等価であることを効率よく求める方法の検討が今後の課題として残されている.

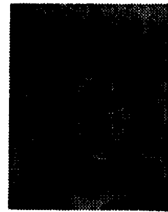
参 考 文 献

1) Darringer, J. A.: The Application of Program Verification Techniques to Hardware Verification, *Proc. 16th DA Conference*, pp. 375-381 (1979).
2) Wojcik, A. S.: Formal Design Verification of Digital Systems, *Proc. 20th DA Conference*, pp. 228-234 (1983).
3) Pitchumani, V. et al.: A Formal Method for Computer Design Verification, *Proc. 19th DA Conference*, pp. 809-814 (1982).
4) 丸山文宏：ハードウェアの機能設計段階における検証，情報処理，Vol. 21, No. 5, pp. 493-503 (1980).
5) Milne, G. J.: A Model for Hardware Description and Verification, *Proc. 21st DA Confer-*

ence, pp. 251-257 (1984).

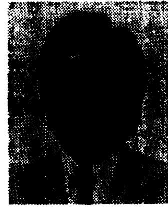
6) Malachi, Y. and Owicki, S. S.: Temporal Specifications of Self-Timed Systems, (Kung, H. T. et al. (ed.)), *VLSI Systems and Computations*, Springer-Verlag, Berlin, pp. 203-212 (1982).
7) 荻原兼一，和田幸一：ハードウェアアルゴリズムの記述と検証，情報処理，Vol. 26, No. 6, pp. 603-612 (1985).
8) Milner, R.: A Calculus of Communicating Systems, (Goos, G. and Hartmanis, J. (ed.)), *Lecture Note in Computer Science 92*, Springer-Verlag, Berlin (1980).
9) INMOS Limited: OCCAM プログラミングマニュアル，啓学出版，東京 (1984).
10) 中村行宏，小栗　清：ハードウェア記述言語とその応用，情報処理，Vol. 25, No. 10, pp. 1033-1040 (1984).
11) Guibas, L. J. and Liang, F. M.: Systolic Stacks, Queues, and Counters, *Proc. 1982 Conference on Advanced Research in VLSI*, pp. 155-164 (1982).
12) Gordon, M.: A Very Simple Model of Sequential Behaviour of nMOS, *Proc. VLSI 81*, pp. 85-94 (1981).
13) Milne, G. J.: Simulation and Verification: Related Techniques for Hardware Analysis, *Proc. CHDL 85*, pp. 404-417 (1985).

(昭和60年10月28日受付)
(昭和61年6月19日採録)



髙原　厚（正会員）

昭和34年生．昭和58年東京工業大学工学部情報工学科卒業．昭和60年同大学院修士課程修了．現在，同博士課程在学中．ハードウェア記述言語，ハードウェア設計検証などに興味をもつ．電子通信学会会員．



南谷　崇（正会員）

昭和21年生．昭和44年東京大学工学部計数工学科卒業．昭和46年同大学院修士課程修了．同年日本電気(株)入社．中央研究所勤務．昭和56年東京工業大学工学部情報工学科助教授，現在に至る．主として論理設計方法論，非同期システム論，計算機の耐故障設計に関する研究に従事．工学博士．著書「PLA の使い方」，「順序機械」(共著)，IEEE，電子通信学会，情報通信学会各会員．