知能と複雑系　115 －5
(1999. 1. 11)

HDLで記述されたハードウェア設計の時相論理による検証

櫟　粛之*　　澤田　宏

NTTコミュニケーション科学研究所

連絡先：櫟　粛之　NTTコミュニケーション科学研究所
〒619-0237　京都府相楽郡精華町光台 2-4
e-mail:　araragi@cslab.kecl.ntt.co.jp

あらまし　本稿では，ハードウェア記述言語(HDL)で記述されたハードウェアの仕様を，その記述から直接，形式的手法を用いて検証する方法を提案する．対象とするHDLは，ハードウェア合成システムPARTHENONで用いられるHDL（SFLと呼ばれる）である．この方法の核となる部分は，SFLで記述されたハードウェアの動作の記述を，その状態遷移関係を表現するブール式に自動変換することである．このブール式が，時相論理CTLのよく知られたモデルチェックアルゴリズムにかけられ，CTL式で表現されたハードウェアの性質が検証される．本稿では，モデルチェックアルゴリズムの検証の結果がSFLの動作セマンティクスに正確に対応するためには，SFLの各動作をどのように論理表現すればよいかを示す．

キーワード　形式的検証，HDL，時相論理，自動合成，PARTHENON

Formal Verification of Hardware Described in HDL Using Temporal Logic

Tadashi Araragi　　Hiroshi Sawada

NTT　Communication Science Laboratories

2-4 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-0237 Japan
e-mail:　araragi@cslab.kecl.ntt.co.jp

Abstract　In this paper, we introduce a method of formally verifying specifications on hardware directly from their descriptions in HDL (Hardware Description Language). The method is focused on SFL, the HDL of the hardware synthesis system PARTHENON. The essential part of the method is transformation of SFL descriptions to the Boolean formulas which represent the relation of state transition of the hardware. In this transformation, several devices are required to create the Boolean formula that behaves precisely in accordance with the semantics of SFL when it is applied to the standard backward model checking algorithm of temporal logic CTL. We discuss the problem in detail.

key words　formal verification, HDL, temporal logic, architecture synthesis, PARTHENON

1　はじめに

ハードウェアなどに代表される有限状態遷移マシンが所望の性質を有するか否かを，論理の枠組みを用いて，自動でしかも完全に検証する方法を一般に形式的検証と呼ぶ．近年，有限状態遷移マシンの状態をBDD（Binary Decision Diagram）を用いて効率良く表現する技術が導入され，形式的検証の中でも時相論理(CTL：Computational Tree Languageなど）のモデルチェック・アルゴリズムによる検証の研究が活発になっている．時相論理に基づく形式的検証を実現した代表的なシステムとしてSMVがある[1]．SMVでは，独自の言語を提供しており，その言語で検証の対象となる有限状態遷移マシンの動作を記述し，CTL式で検証したい仕様を表現する．SMVは，与えられた遷移マシンの動作の記述をもとに，その動作の遷移関係をブール式（BDD）に変換し，得られた式とCTL式をモデルチェックのアルゴリズムにかける．

本研究では，この検証手法を直接ハードウェア合成システムに埋め込むことを目的とする．したがって研究の主眼は，合成システムが提供する言語を対象とし，そこで記述されたハードウェアを，その言語のセマンティクスに基づいて，検証アルゴリズムに正しく反映される式に変換することにある．ここでは対象とするハードウェア合成システムとしてPARTHENON [2]を考える．PARTHENONでは，SFL(Structured Function description Language)と呼ばれる，特有のハードウェア記述言語（Hardware Description Language：以下HDLと略す）を用いる．一般的なHDLとしてはVHDLやVerilog-HDLが代表的であるが，これらの言語のセマンティクスは複雑で状態遷移を表す論理式を得ることが困難である．これに対しSFLは，表現力は制限されるものの，状態遷移のセマンティクスを強く反映しており，自動合成のみならず，形式的検証にも親和性が高い．

2　時相論理を用いた形式的検証

2.1　時相論理と形式的検証の関係

時相論理は，クリプケモデルと呼ばれる，有向グラフで各ノードに状態の真偽が定まっている数学的構造をモデルとする論理である．一方ハードウェアに代表される有限状態遷移マシンの動作は，状態遷移図，即ち，辺にラベルのついた有向グラフで各ノードに状態の真偽が定まっている数学的構造で表現される．そして，図1にみられるように，状態遷移図は，クリプケモデルに変換できる（例えば，状態遷移 $s1 \xrightarrow{a} s2$ は，クリプケモデルでは，s1, a → s2,a, s1, a → s2, b の2つの遷移で表現される）．したがって，有限マシンの動作の検証は，その動作をクリプケモデルで表現し，調べたい性質を時相論理式で表現して，その論理式がそのモデルを満たすか否かのモデルチェック手続きで可能になる[3].

実際のモデルチェックの処理では，状態遷移を表すクリプケモデルはブール式（一般にBDD形式を使う）で表現され，モデルチェックのアルゴリズムは，論理式の論理演算で行われる．特に逆像計算と呼ばれる状態遷移を逆にたどる論理演算を用いて，検証したい仕様を満たすノードを全て求める計算が行われる．たとえば，「いつかはpが成り立つ」という性質は，pが成り立つ全てのノードの集合（論理式では単にpで表される）から遷移を逆にたどり，その仮定で通過した全てのノードの集合が「いつかはpが成り立つ」を満たすノードとなる．初期状態を表す式が計算結果の式を論理的に含意（imply）していれば，調べている性質が満たされることになる．逆像計算はある種の後ろ向き推論に相当する．

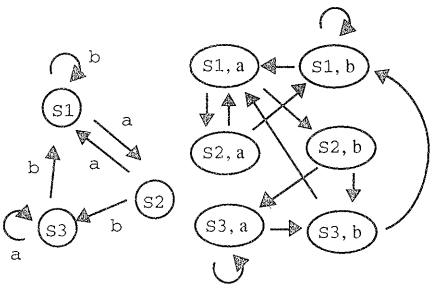

図1：状態遷移図とクリプケモデル

2.2　時相論理CTL

時相論理CTLの簡略な説明を与える．CTLは様相論理の一つで，AX，EX，AG，AF，EG，EF，A(-U-)，E(-U-)などの様相記号を持つ．

クリプケモデル上では，Xは次のステップのノードを表す．Aは「全てのパス」を表し，Eは「あるパス」を表す．Gは「パス上のどこでも」を表し，Fは「パス上のどこかで」を表す．最後に，Uは「パス上..が成り立つまでは..が成り立つ」を表す．例えば，AG ϕ は，着目しているノードから，

どんなパスを選んでもそのパス上の全てのノードでϕが成立することを意味する．

遷移マシンに関する仕様はこのCTLの論理式を使って表現される．クリプケモデルが有限の場合，CTL式を満たすノードの集合を求める決定的なアルゴリズムが得られている．

3 SFL

3.1 SFLの構造

SFLの記述は，モジュール（*module*）を基本単位とする．モジュールには，データの入力・出力端子（*input*, *output*）及びモジュールの機能を呼びだす起動ボタンである制御入力・出力端子（*instrin*, *instrout*）がインターフェース部分に配置される．また，内部には，レジスタ（*reg*），メモリ（*mem*），内部でデータをやりとりするワイヤ（*sel*）と，別のモジュールのインスタンス（サブモジュール）が配置される．制御入力端子は，そのモジュールの機能を外から利用する場合に使われる．即ち，データ入力端子にデータを供給し，制御入力端子を起動し，結果をデータ出力端子から得る．逆に制御出力端子は，モジュールの外側の，スーパーモジュールの機能を利用する場合に使われる．即ち，データ出力端子にデータを供給し，制御出力端子を起動し，結果をデータ入力端子から得る．

制御端子の起動時に供給されるデータを，どのデータ線に渡すかを指定するために*instr_arg*宣言が行われる．"instr_arg A(i1, i2);"と宣言され，"A(d1, d2)"という形で，制御端子*A*が起動されたときデータ d1，d2 は各々データ線 i1，i2 に供給される．

SFLはオブジェクト指向言語に類似した構造を持つ．Javaでいえば，モジュールの定義はクラスの定義に対応し，サブモジュールの指定はクラスのインスタンス生成に相当する．ただしメソッドの呼び出しは配置上で親子関係にあるモジュールの間だけに限られる．

3.2 SFLの動作

上記の構造から分かるように，各モジュールでは，その制御入力端子が外から起動されたときの動作と，そのモジュールが含むサブモジュールの制御出力端子がそのサブモジュールの内部から起動されたときの動作を定義しなければならない（図２）．また，電源が投入された時に，モジュール内のレジスタなどの初期状態を設定する動作も必要になる．本稿では便宜的に前者を制御端子動作，後者を初期動作と呼ぶことにする．

SFLでの動作記述の中心となるのがステージである．これはモジュールの状態遷移動作を記述する基本の単位である．ステージは有限状態遷移の直接的な記述で，複数のステートを持ち，各ステートごとに，そのステートで行う動作が記述されている．

電源投入時にステージのステートは，*first_state*宣言されたステート値に設定される．ステージが起動されたとき，ステートは前回終了したステート値からはじまる．電源投入後はじめての起動であれば*first_state*宣言されたステート値である．ステージ起動中は，現在いるステート値に書かれた動作を実行する．ステージが実行する動作の中には，goto動作とfinish動作がある．"*goto ステート値;*" は次のクロックでステージのステートはステート値に移ることを表す．"*finish*" は次のクロックで，ステージの起動が終了することを表す．

一つのモジュールにステージは複数あってもよいが，共通のクロックを用いる．即ち，同期している．さらに，全てのモジュール間でステージの動作は同期している．

動作の定義は，このステージを中心に，制御出力端子の機能や，サブモジュールの制御入力端子の機能およびレジスタやメモリという記憶媒体を利用しながら定義される．各種動作については4.2でもう一度詳しく述べる．

ステージがモジュール内での動作を表し，制御端子がモジュールをまたがる動作の起動を表す．ただしJavaとは違って，直接呼び出し可能な他のモジュールの機能は，スーパーモジュールか，サブモジュール，あるいは自分自身のものに限られる．

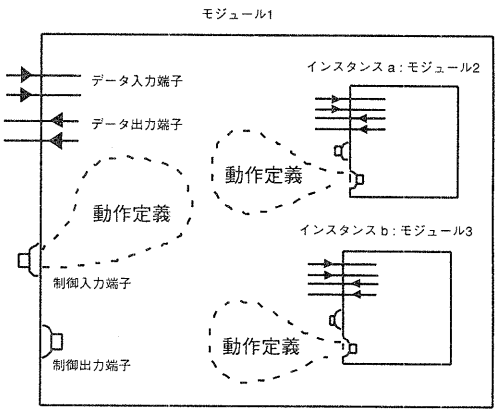

図２：SFL の構造

4　論理式への変換

SFLはモジュールを単位とした記述である．各モジュールにはそのモジュールが提供する機能，即ち，そのモジュールの動作が定義される．その動作定義においては，そのモジュールのサブモジュールや，スーパーモジュールの提供する機能（制御端子）が利用できる．これによってモジュールごとの動作が他のモジュールの動作とつながり，全体の動作を形成する．

この全体の動作を表現する論理式を構成するために，まず，SFLの記述からモジュールの物理的（木）構造を抽出する．次に，モジュールごとにそのモジュールが提供する動作の定義を論理式に変換する．そして上記の木構造を利用して全体の論理式を組み立てる．最後に，モデルチェックアルゴリズムで正しくSFLのセマンティクスが反映されるように，この論理式を変形する．

4.1　モジュールの構造木の構成

各モジュールの記述において，サブモジュールの宣言が，"サブモジュール名　サブモジュールのインスタンス名；" の形で行われている．この情報をもとに全体のモジュールの木構造が得られる．この木は，一番外側にあるモジュールのインスタンスの名前をデフォルトの"*Top*"と表し，<インスタンス名：モジュール名> という対の列がノードを表す．この列を以後 インスタンスIDと呼ぶことにする．インスタンスIDはSFLで記述されたシステムで使われるモジュールのインスタンスのIDを表している（図3）．

4.2　モジュールごとの論理式への変換

各モジュールの記述で，直接，論理式への変換の対象となるのが，ステージ動作の定義，制御端子動作の定義，初期動作の定義の３つである．

これらの動作は，以下に説明する条件判定と基本動作を用いて「条件判定がなりたつならば基本動作を行う」という形の動作の連言に書き直せる．この事実に基づいて上記の動作を，条件判定式→基本動作式の連言に書き換える．まずは，基本動作，条件判定とそれに対応する論理式を説明する．

[基本動作]
基本動作は以下にあげる５つのものである．

・レジスタへの値の代入：
　これは，
　　レジスタ名^= 値
　という式で表現する．
・ステージが起動状態でのステートの移動：
　これは，
　　ステージ名.タスク名.state^ = state名
　という式で表現する．
・ステージの起動(*generate*)：
　これは，
　　ステージ名.タスク名.generate
　という変数の式で表現する．
・ステージの終了(*finish*)
　これは，
　　ステージ名.タスク名.finish
　という式で表現する．
・制御端子の起動
　これは，
　　制御端子名
　という変数の式で表現する．

値は，ワイヤ，レジスタのブール式で表される．ここで=と書かれているのは，論理記号の同値(<->)を表している．また，レジスタがビット幅を持つときには，"レジスタ名^ = 値" は，正確には，"(レジスタ名0^ = 値0)∧(レジスタ名1^ = 値1)∧. . ." である．以下，ビット幅をもつ代入の扱いは上記と同じとする．

"変数名^" の「^」は，"変数名" に対応する，次のクロックでの変数を表す．したがって，"変数名" と "変数名^" は別の変数である．

制御端子の起動及びステージの起動では，一般に

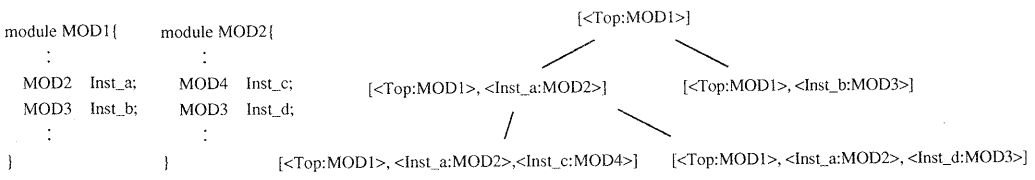

図３：モジュールのインスタンスの構造木

入力が与えられる．*stage_arg* 及び*instr_arg* の宣言でデータ端子と引数の位置の対応が与えられるので，入力をもった制御端子の起動，およびステージの起動に対し，データ端子と入力値の対応が得られる．この対応に基づいて，データ端子名=値　という形の式を作り，その連言をgenerate変数，制御端子名変数に対応づける．例えば，"instruct_arg 制御端子名(w1, w2);" という宣言が与えられていて，"制御端子名(v1, v2)" の起動が行われる場合，制御端子名には，"w1=v1∧w2=v2" の式が対応づけられる．

[条件判定]
条件判定は以下の基本判定のブール結合である．
・データ端子の値の判定
　これを表す式は，
　　データ端子の変数名
　または
　　データ端子の変数名=値
　である．
・レジスタの値の判定
　これを表す式は，
　　レジスタの変数名
　または
　　レジスタの変数名=値
　である．
・ステートの値の判定
　これを表す式は，
　　モジュール名.タスク名.state = state名
　である．

動作の定義は，一般に*par*, *any*, *alt*, *if* など結合子を用いて複合的なものになっている．しかしSFLで定められているこれらの結合子の解釈から，動作は，基本条件判定式のブール式→基本動作式　の連言に書き換えられることが容易にわかる（本稿ではこの書き換え形の説明は省略する）．このような形の式に書き換えられる理由は，SFLが決定的な動作の記述しか許さないことと，同期システムであることによる．

4.3　全体の論理式の構成

（1）モジュールに付随する動作式の組み合わせ
モジュール木のノードを n1, n2, . . とする．ノード n に対応するモジュールのステージ動作式を S1, S2, . . , 初期動作式をIとする．ノードのインスタンスIDを，S1, S2, . ., I に出現する各変数にprefixする．これを S1', S2', . . ,I' とする．このとき連言 S1'∧S2'∧. . .∧(init→I') を作り，これを Mn とする．この Mn の全てのノードににについての連言 Mn1∧Mn2∧. . . をとる．これを R と書く．

（2）非状態変数の消去
(2-1) 制御端子名の消去
式Rに出現する制御端子名をその動作式で置き換える．ただし，その制御端子名にprefixされたインスタンスIDを p とすると，動作式に出現する変数に以下の規則で作られるインスタンスID p' をprefixする．まず，制御端子が出力端子の場合，pの最後の元を除いたものを p' とする．入力端子の場合，制御端子名には，サブモジュール名が含まれる．これをnとし，そのモジュールをmとして，p' を p*<n:m> とする．内部端子の場合は，p' は p そのものである．また，その制御端子名に対応するデータ端子・値対応の式が対応づけられている場合には，その式も連言に加える．
(2-2) データ端子名の消去
上記の操作で更新された式Rにおいてデータ端子・値対応の処理で加えられた「データ端子名=式」の連言因子があった場合，これを利用して「データ端子名」の出現を「式」で置き換えて，データ端子名を消去する．ここでも，消去対象のデータ端子名にインスタンスID p がprefixされていたら，置き換える式の全ての変数の出現に p をprefixする．
　上記の操作(2-1)，(2-2)で，[<Top:...>] のモジュールインスタンス以外のデータ端子名，制御端子名は全て消去される．

4.4　式の変形

4.3 の構成で得られた式Rは，適当な変形（例えば基本動作式および基本判定式をアトムと見た連言標準形への変換）により以下の形の式に同値変換される．

$$
\begin{aligned}
&(f1(\vec{s}) \rightarrow h1)\\
&\wedge(f2(\vec{s}) \rightarrow h2)\\
&\quad\vdots\\
&\wedge g1(\vec{s})\\
&\wedge g2(\vec{s})\\
&\quad\vdots
\end{aligned}
$$

ここで $\vec{s}$ は R に出現する全ての基本判定式のベク

トルであり，hi は制御端子の起動以外の基本動作式のどれかである．fi($\vec{s}$)，gi($\vec{s}$) はともに，$\vec{s}$ 中の基本判定式のブール結合である．また，(B→A)∧(C→A) と (B∨C→A) は同値なので，hi はどれも異なると仮定してよい．

以下では，検証における逆像計算の後ろ向き推論を行う場合に，SFLのセマンティクスに合致したふるまいを実現するために式Rを書き換える．

（1）レジスタ代入式の２値化

hi がレジスタへの値の代入の式，即ち，"レジスタ名^= 値" の形のとき，

(fi($\vec{s}$) → hi)

なる連言因子を

(fi($\vec{s}$)∧値 → レジスタ名^)

∧(fi($\vec{s}$)∧¬値 → ¬レジスタ名^)

に書き換える．全て書き換えたら，ここでもう一度 head部の一意化を行う．

（2）レジスタ代入式のデフォルト動作

A → レジスタ名

B →¬レジスタ名

の連言因子に対して，各々

(¬A∧¬B∧レジスタ名)∨A→レジスタ名^

(¬A∧¬B∧¬レジスタ名)∨B→¬レジスタ名^

に書き換える．新しく加わった選言因子は，レジスタの値の変化がなかった場合は次のサイクルでレジスタの値が，現在のものと同じであることを表している．

（3）ステージ式のデフォルト動作

今，あるステージ stage，そのタスク task に対し，変数 stage.task.state を St と略記したとき，

A1 → St^=st1

A2 → St^=st2

:

をstage，taskの状態遷移に関する全ての連言因子とし，変数 stage.task.run を Run と略記したとき，

(¬Run∧St = st1)∨(Run∧A1) → St^ = st1

(¬Run∧St = st1)∨(Run∧A2) → St^ = st 2

:

に書き換える．ここでも新しく加わった選言因子が，ステージが起動していない場合，St の値が変わらないことを表している．さらに，ある Ai の中に init が選言因子として出現していたら，Ai からその選言因子を除いたものを Ai' としたとき，連言因子の (Run∧Ai) の部分を (Run∧Ai')∨init に書き換える．

次に，

G → stage.task.generate

F → stage.task.finish

の形の連言因子に対し，これらを

G∨(¬F∧Run) → Run^

(F∧Run)∨(¬G∧¬Run) → ¬Run^

に書き換える．これにより，ステージの起動・不起動が表現される．

（4）最後に連言因子 (fi($\vec{s}$) → hi) を後ろ向き推論に合わせるために (fi($\vec{s}$) ← hi) に書き換える．

5 適用例

以下はSFLの記述例である[4]．これは，サブモジュールを持たないモジュールで，alarm_stgと呼ばれるステージを１つ持つ．

```
module alarm_4 {
    instrout left_lamp ;      ←  制御出力端子の宣言
    instrout right_lamp ;
    instrout horn ;
    instrin approach;      ←  制御入力端子の宣言
    stage_name alarm_stg {      ←  ステージの宣言
        task alarm_task();      制御入力端子の動作定義
    }                                    ↓
    instruct approach generate alarm_stg.alarm_task();
    stage alarm_stg {              ←  ステージの動作定義
        state_name st1, st2;  ←  ステートの宣言
        first_state st1;
        state st1 par{      ←  ステートの動作定義
          left_lamp();
          horn();
          goto st2;
        }
        state st2 par {
          right_lamp ();
          horn ();
          goto st1;
          finish ;  ---(1)
        }
    }
}
```

この例に対し，(1)の行がある場合と，ない場合に

ついて，上記の変換手続きの適用を考えると，変換結果は以下のようになる．ただし各行は連言で結ばれているとし，インスタンスIDやステージ名，タスク名はユニークなので省略している．

state=s1∨approach∨(run∧(state=s2)) ← state^ = s1
state=s2∨(run∧(state=s1)) ← state^ = s2
(1)の行がある場合
approach∨(¬(state=s2)∧run) ← run^
((state=s2)∧run)∨(¬approach∧¬run) ← ¬run^
(1)の行がない場合
approach∨run← run^
¬approach∧¬run ← ¬run^

(1) の行がある場合，
M ⊨approach⊃EFAG(state=s1)
即ち，「ある時間から先，状態はs1に落ち着く」ことが自動検証される．
(1) の行がない場合，
M ⊨approach⊃ EFAG(((state=s1)∧EX(state=s2))
∨ ((state=s2)∧EX(state=s1)))
即ち，「ある時間から先，状態はs1とs2を交互に繰り返し，停止しない」ことが自動検証される．

図4にSFLの記述例をもう一つ示す[2]．記述されたシステムは，タイマーでありSTART制御端子を押すとINITに与えられた数（ここでは7～0）のカウントダウンを行う．RESET制御端子が押されると次のクロックから，カウントダウンを停止する．また，カウントダウンで0までくると，カウントを終了したことを知らせる信号が，STARTあるいはRESETボタンが押されるまでEXPIREに出力され続ける．システムの構造は，TIMERモジュールが一番外側にあり，その中にサブモジュールDECR3がある．このサブモジュールのインスタンス名はDECRである．DECR3は，与えられた数を1減らす機能を持ち，ステージは持たない．

モジュール木は[[<Top:TIMER>], [<Top:TIMER>, <DECR:DECR3>]]となる．ただし，DECR3はステージもレジスタも持たないので，論理式に現れるprefixは[<Top:TIMER>]のみである（ここではこのprefixも，またステージ名，タスク名も省略している）．TIMERは一つのステージを持ち，そのステージの取りうるステート値はDOWNとASSERTの2つである．またレジスタREMAINEDを1つ持ち，ここにカウントダウンした値が記憶される．変換された式からわかるように，REMAINEDの変化は，DECRを通して，TIMERのステートに依存している．

5 おわりに

本稿では，SFLを対象に，その記述から直接，仕様検証のための時相論理のモデルチェックアルゴリズムに適用できるブール式を構成する方法を示した．時相論理のモデルチェックでは，状態を表現するBDDの変数順序が計算効率に大きく影響する．SFLは構造化されたセマンティクスを持つので，その構造を利用して良い変数順序を決定することを次の課題とする．

謝辞

この研究のきっかけを与えて下さったNTT NWサービスシステム研究所，石川啓二氏，また，研究の立ち上げに際して，いろいろなアドバイスを下さったNTT光ネットワーク研究所，高原厚氏，湊真一氏に感謝します．

また，本研究をご支援下さった，NTTコミュニケーション科学研究所 東倉洋一所長，服部文夫計算機科学研究部長，小幕潔主幹員，名古屋彰主幹員に感謝します．

参考文献

[1] McMillan. K. L. : *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
[2] 小栗 清, 名古屋 彰, 野村 亮, 雪下 充輝 : はじめてのPARTHENON, CQ出版社, 1994.
[3] 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, 情報処理学会誌, Vol.35, No.8, pp.710-718, 1994.
[4] 中島 彰, 野村 亮, 小栗 清, 名古屋 彰, 桑原 敏 : ハードウェア記述言語によるASIC設計講座,トランジスタ技術, 1994年6月号 ～1995年5月号.

```
declare TIMER {
    instr_arg  START( INIT ) ;
}

module TIMER {
    instrin    START;
    instrin     RESET ;
    instrout   EXPIRE ;
    input      INIT<3>     ;
    reg        REMAINED<3> ;
    DECR3      DECR ;

    stage_name MAIN { task RUN( REMAINED ) ; }
    instruct   START generate MAIN.RUN( INIT ) ;

    stage MAIN {
        state_name  DOWN, ASSERT ;
        first_state DOWN ;

        state DOWN  any{
        RESET ¦ START : finish ;
        else          : par{
        REMAINED := DECR.ENABLE( REMAINED ).OUT ;
        if( DECR.OUT == 0x00 ) goto ASSERT ;
            }
        }
        state ASSERT any{
        RESET ¦ START : par{ goto DOWN ; finish ; }
            else          : EXPIRE() ;
        }
    }
}

declare DECR3 {
    instr_arg  ENABLE( IN ) ;
}

module DECR3 {
    input     IN<3> ;
    output    OUT<3> ;
    sel_v     C<3>  ;
    instrin   ENABLE ;

    instruct  ENABLE par{
    C =  /¦( IN<1:0> )
            ¦¦ IN<0>
            ¦¦ 0b0 ;
    OUT = ^( IN @ C ) ;
    }
}
```

図4：変換の例

論理式

```
/* - - - - レジスタの変化 - - - - */
(¬((st = DOWN)∧(¬RESET∧¬START))∧¬START∧REMAINED2)
∨((st = DOWN)∧(¬RESET∧¬START)
    ∧(REMAINED2 = (REMAINED1∨REMAINE0)))
∨(START∧INIT2)
← REMAINED2^

(¬((st = DOWN)∧(¬RESET∧¬START))∧¬START∧¬REMAINED2)
∨((st = DOWN)∧(¬RESET∧¬START)
    ∧¬(REMAINED2 = (REMAINED1∨REMAINE0)))
∨(START∧¬INIT2)
← ¬REMAINED2^

(¬((st = DOWN)∧(¬RESET∧¬START))∧¬START∧REMAINED1)
∨((st = DOWN)∧(¬RESET∧¬START)∧(REMAINED1 = REMAINE0))
∨(START∧INIT1)
← REMAINED1^

(¬((st = DOWN)∧(¬RESET∧¬START))∧¬START∧¬REMAINED1)
∨((st = DOWN)∧(¬RESET∧¬START)∧¬(REMAINED1 = REMAINE0))
∨(START∧¬INIT1)
← ¬REMAINED1^

(¬((st = DOWN)∧(¬RESET∧¬START))∧¬START∧REMAINED0)
∨((st = DOWN)∧(¬RESET∧¬START)∧¬REMAINE0)
∨(START∧INIT0)
← REMAINED0^

(¬((st = DOWN)∧(¬RESET∧¬START))∧¬START∧¬REMAINED0)
∨((st = DOWN)∧(¬RESET∧¬START)∧¬(¬REMAINE0))
∨(START∧¬INIT0)
← ¬REMAINED0^

/* - - - - ステージのステート遷移 - - - - */
(run∧(st = DOWN)∧(¬RESET∧¬START)
    ∧¬(REMAINED2 = (REMAINED1∨REMAINE0))
    ∧¬(REMAINED1 = REMAINE0)∧¬(¬REMAINED1))
∨(¬run∧(st = ASSERT))
←  st^ = ASSERT

(run∧(st = ASSERT)∧(RESET∨START))
∨(¬run∧(st = DOWN)
∨init
← st^ = DOWN

/* - - - - ステージのrunの変化 - - - - */
START
∨(¬(((st = DOWN)∨(st = ASSERT))∧(RESET∨START))∧run)
← run^

(((st = DOWN)∨(st = ASSERT))∧(RESET∨START)∧run)
∨(¬START∧¬run)
← ¬run^
```