社団法人　情報処理学会　研究報告
IPSJ SIG Technical Reports

2005－ＳＬＤＭ－121（32）
2005／10／21

モニタベース形式検証のための入力制約を考慮したモニタ回路生成手法

垣内　洋介†　北嶋　暁††　浜口　清治†　柏原　敏伸†

† 大阪大学大学院情報科学研究科
†† 大阪電気通信大学総合情報学部メディアコンピュータシステム学科
E-mail: †{kakiuti,hama,kashi}@ist.osaka-u.ac.jp, ††kitajima@sldlab.osakac.ac.jp

あらまし　本論文では，ハードウェアインターフェース仕様全体をより包括的に検証することを目指し，正規表現ベースの仕様記述言語からモニタ回路を生成する手法を示す．モニタ回路の生成は自動的に行われるため，複雑な仕様に対するモニタ回路であっても人手で設計する場合に比べ，誤りの混入を避けることができる．また，モジュール単位の検証の際，通常では入力制約を付加する必要があるが，本論文では仕様記述から，入力制約を自動的に抽出して，モニタ回路に直接反映させる手法について述べる．実験では設計例としてバスプロトコルである AMBA AHB に従って設計された回路を取り上げ，実際に仕様を記述してモニタを生成し，形式検証を行った結果を示す．

キーワード　形式検証，モニタベース検証，AMBA，モニタ回路生成

A Monitor Generation Method for Formal Monitor-based Verification Considering Input Constraints

Yosuke KAKIUCHI†, Akira KITAJIMA††, Kiyoharu HAMAGUCHI†, and Toshinobu KASHIWABARA†

† Graduate School of Information Science and Technology, Osaka University
†† Faculty of Information Science and Technology, Osaka Electro-Communication University
E-mail: †{kakiuti,hama,kashi}@ist.osaka-u.ac.jp, ††kitajima@sldlab.osakac.ac.jp

Abstract　In order to verify hardware module interfaces, various verification methods have been proposed. This paper focuses on formal monitor-based verification of module interfaces. In our method, first, we describe specifications of module interfaces in a language based on regular expressions, then construct behavior models from the description. Finally, we generate a monitor circuit. When we verify module interfaces formally, some input constraints are usually required. We show how to generate monitors including input constraints automatically from specifications. In verification experiments, we show that modules that comply AMBA AHB bus protocol can be verified by monitors formally.

Key words　Formal Verification, Monitor-based Verification, AMBA, Monitor Circuit Generation

1. はじめに

システム LSI 設計における工数の増大に伴い，既存の設計資産 (Intellectual Property; IP) をモジュール単位で再利用し，工数を削減する手法が取り入れられてきている．IP としてモジュールを再利用する場合，そのモジュールのインターフェース仕様を明確化し，機能検証を行い，システム内の他の部分と正しく通信が行えることの保証が最も重要な要件の一つとなる．

検証手法として最も一般的な方法にシミュレーションがあるが，大規模になるにつれて検証の網羅性が問題となり，コーナーケースと呼ばれる発見しにくい誤りが残存しやすくなる．

これに対して，設計仕様を形式的に記述し，計算機によって設計誤りを自動的に発見する形式検証手法がハードウェアモジュールのインターフェース仕様検証に利用されている．検証手法の一つにアサーションを用いた検証がある．アサーションとは検証対象が満たすべき性質のことであり，通常は PSL(Property Specification Language) や SVA(SystemVerilog Assertion) などのアサーション記述言語で記述し，検証時にはアサーションが満足されるかどうかが自動的にチェックされる．アサーションベース検証ではアサーションを複数記述することで回路の正しさを調べるが，記述したアサーション集合が仕様を全てカバーしているかを判定することが困難である．その結果，コーナー

ケースの誤りを発見できない可能性が残る．一方，モニタを用いてバスプロトコルを検証する手法も提案されている[1]．モニタとはモジュールのインターフェース部を監視し，インターフェースが仕様を満足しているかを監視する回路のことである．検証においてモニタが誤りを検出すると，エラー信号を出力する．検証ツールはこのエラー信号が出力されないかどうかを調べる．しかし，従来のモニタを用いた検証においては，仕様からモニタを回路として新たに設計せねばならず，その段階で仕様の読み違いなどから誤りが混入する可能性がある．

そこで，本論文では仕様を自然言語ではなく正規表現ベースの仕様記述言語で形式的に記述し，その仕様記述からモニタを自動生成する手法について述べる．仕様記述言語には日立・富士通・富士通研究所が提案するCWL[6]を用いる．CWLではモジュールのインターフェース仕様全体を一つにまとめて記述するため，アサーションと比べて，より包括的な検証に向いていると考えられる．また，本論文では，大規模システム中のモジュール単体の形式検証に利用できるよう，モジュールへの入力制約を考慮したモニタを生成する手法についても提案する．これはシステム全体に形式検証を適用することが規模的に困難な場合や，初期段階の検証において，モジュール単体で検証を行う際に必要となる入力制約をモニタに反映させるものである．CWL仕様記述がインターフェース仕様の全体を記述しているので，仕様から入力制約を自動的に抽出できる．検証を行う際に，入力を制約する回路を別途設計する必要がないため，手間を軽減することが可能である．

実験ではAMBA AHB[5]に従ってインターフェースが設計されたモジュールについて検証を行う．AMBA AHBはARM社の実用的なシステムLSI向けのバスプロトコルであり，既に様々な検証の試みがなされているが[2][3]，いずれも仕様の一部分を取り出して，それに対応した状態機械を手動で記述し，検証ツールに入力するというものであり，仕様全体を記述しようとすると状態機械が大きくなり記述が困難になる．CWLの記述では状態機械を直接記述するよりもわかりやすく仕様が記述でき，モニタはそこから自動生成されるため，モニタ生成の過程では誤りの混入がない．本論文ではAMBA AHBのCWL仕様記述からモニタを生成し，それを検証対象となるモジュールに接続して，モジュールがAMBA AHBの仕様を満たしているかを検証する．その結果，モニタ自動生成手法が実設計例に対しても適用できることを示す．

2. 動作モデルの構築

以下では，正規表現ベースの仕様記述言語から動作モデルを構築する方法について説明する．図1に形式検証のためのモニタ生成の全体の流れを示す．

2.1 正規表現ベースの仕様記述

CWLでは，モジュールのインターフェースに現れる信号値の組を一つの記号として定義し，インターフェース仕様全体を定義した記号を使った正規表現で表す．図2にその例を示す．図では，例えば $(SYNC,\ ACK,\ \_ENABLE,\ RD/WR,\ DATA) = (0, 0, -, -, -)$ (− はドントケア) という値の組を記号 I として

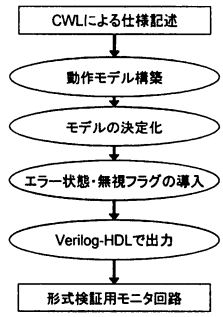

図1 仕様からモニタが自動生成されるまでの流れ

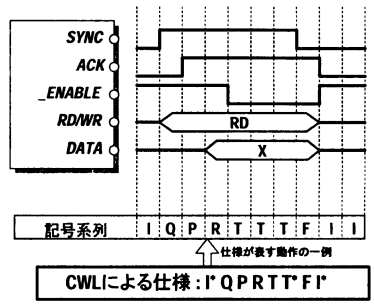

図2 CWLによる仕様記述例

定義している．CWLでは正規表現が表す動作の一例が波形に対応しており，0回以上の繰り返しである「*」などが含まれるため，一般に1つの正規表現が複数の動作パターンを表現している．

CWLは独自の表記法を導入しており「〜が起こった後，いつか必ず … が起こる」「〜と … が同時に並行して起こる」といった同期や並列動作に関する記述も可能である．また，ユーザは変数を宣言して正規表現中の特定の箇所でその変数に値を代入したり，変数を用いて条件を記述し，ある正規表現は条件が成り立つときのみ有効であるようにすることができる．なお，本手法では変数はブール変数もしくはその配列と仮定する．

2.2 動作モデル

次にCWLで記述された仕様から動作モデルを構築する．動作モデルの全体図を図3に示す．動作モデルは拡張を加えた有限状態機械 (Finite State Machine; FSM) とキューが複数並列に動作する形になる．

動作モデルのうち，有限状態機械を6つ組 $M = (S, V, I, f, g, S_0)$ で定義する．S は状態集合であり，V はユーザ定義変数の集合である．I はインターフェース信号の集合で各要素は入力もしくは出力どちらかの属性を持つ．$f : S \times 2^V \times 2^I \rightarrow 2^S$ は非決定的な状態遷移関数であり，$g : S \times 2^V \times 2^I \rightarrow 2^V$ はユーザ定義変数への代入の遷移関数である．S_0 は初期状態

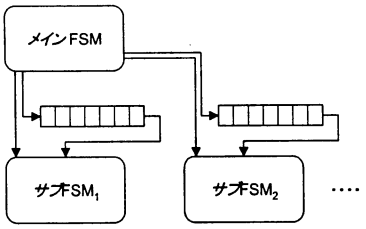

図 3　動作モデル

集合である．f と g は正規表現とユーザ定義変数を用いた条件式および代入文から決まる．仕様のうち，正規表現の部分については標準的な方法で非決定的な有限状態機械へと変換する．ここでは，正規表現から状態機械を得る方法については割愛する．この段階で，モジュールの入出力信号の集合が I になり，V は仕様でユーザが定義したブール変数の集合になる．遷移関数 f, g は正規表現と仕様中の代入文から決定される．

例えば，「A が実行された後 y に 1 を代入し，その後 x が 0 の場合のみ B を実行する」という仕様，

$$A\{y \leftarrow 1\}\{x == 0\}B$$

を考える．CWL による仕様記述では正規表現の後に代入文を書くと，その正規表現の実行が終わる際に代入が行われる．また，正規表現の前に条件式を書くことで，その条件が成り立つ場合にのみ正規表現が実行される．ここでは，インターフェースには Q と R という端子が存在し，記号 A は $(Q, R) = (1, 0)$，記号 B は $(Q, R) = (1, 1)$ と定義されているとする．この仕様からは図 4 のようなモデルが構成される．

$$\begin{aligned}
S &= \{q_0, q_1, q_2\} \\
V &= \{x, y\} \\
I &= \{Q, R\} \\
f &= \{(q_0, \{\}, \{Q\}, \{q_1\}), (q_0, \{x\}, \{Q\}, \{q_1\}), \\
&\quad (q_0, \{y\}, \{Q\}, \{q_1\}), (q_0, \{x, y\}, \{Q\}, \{q_1\}), \\
&\quad (q_1, \{x\}, \{Q, R\}, \{q_2\}), \\
&\quad (q_1, \{x, y\}, \{Q, R\}, \{q_2\})\} \\
g &= \{(q_0, \{\}, \{Q\}, \{y\}), \\
&\quad (q_0, \{x\}, \{Q\}, \{x, y\}) \ldots\} \\
S_0 &= q_0
\end{aligned}$$

ただし，ユーザ定義変数は代入が行われない場合は値を保持するので，g についてはこれ以外の場合について，値が変わらないという遷移が存在するが，ここでは明らかなので省略する．f において，例えば $(q_0, \{x, y\}, \{Q\}, \{q_1\})$ は，「現状態が q_0 で $(x, y) == (1, 1)$ であり，(Q, R)==(1, 0) である場合，次状態は q_1 である」ということを表している．

このように，動作モデルを形式化し，通常の有限オートマトンの各状態において，V と I の各要素がそれぞれ 0 と 1 どちらを取ればどの状態に遷移するかを決定した．しかし，直観的には図 4 のように遷移枝に条件となる論理式が付いていると考える方がわかりやすい．そこで，次状態関数 f において，$V \cup I$ の要素を用いた論理式と $V \cup I$ の部分集合を対応させることを考える．まず $V \cup I$ の部分集合を考える．部分集合に含まれる変数は 1 であり，含まれない変数は 0 と考えるとブーリアンベクトルを表すことができる．一方，論理式ではその論理式を真にするような各変数への真偽割り当てによりブーリアンベクトルの集合を表現できる．よって，仕様に書かれている論理式が，そのまま遷移枝に付加されていると定式化しても問題はない．よって，以降では状態 q において，条件 c （c は論理式）で状態 $\{t_1, t_2, \ldots, t_n\}$ に非決定的に遷移することを $f(q, c) = \{t_1, t_2, \ldots, t_n\}$ とも表し，c を遷移条件と呼ぶ．

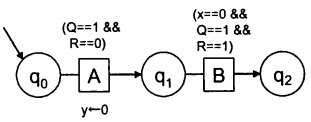

図 4　$A\{y \leftarrow 1\}\{x == 0\}B$ に対するモデル

次に代入文はユーザ定義変数に対するブーリアンベクトル値の代入であり，ベクトル値は $V \cup I$ の部分集合に対応する．つまり，その部分集合が g における次状態と思えばよい．よって代入文も遷移枝に付加されており，遷移する際に代入が起こると考えて問題ない．しかし，f が非決定性を許している一方で，ユーザ定義変数に対する遷移 g は決定的である．つまり，ある状態 q において，複数の遷移条件 $c_1, c_2, \ldots c_n$ が成立し，そのいずれかで同じ変数へ異なる値を代入しようとするとき，代入の衝突が起こる．また，異なる状態 q_a, q_b から t_a, t_b への遷移 $(q_a, c, t_a), (q_a, c, t_b)$ に代入文 $v \leftarrow z_a, v \leftarrow z_b$ がそれぞれ付加されている場合についても，q_a, q_b に非決定的に遷移した後に c が成り立てば，代入の衝突が起こる．実装ではこのような代入の衝突はあらかじめ仕様には存在しないものとしている．

仕様に並列動作や分割トランザクション，パイプラインの記述が存在する場合は，いくつかの状態機械が並列した形の有限状態機械のネットワークとなる．分割トランザクションとは，転送が一度保留されるなどして，二つ以上に分かれて実行されるようなデータ転送を言う．ただし，その場合は同期の中心となるメインの状態機械が 1 つと，それ以外の状態機械が複数という形態を取る．この場合，「〜が起こった後，次サイクルに ‥ が起こる」という形の同期は大域変数を用いて実現され，仕様に「〜が起こった後，いつか必ず ‥ が起こる」という分割トランザクションはプロセスキューを用いてその動作をモデルに反映する．例えば，正規表現 A, B に対し「A が起こった後，いつか必ず B が起こる」という動作には，A および B に対する状態機械 M_A, M_B をそれぞれ生成し，プロセスキューを 1 つ用意する．M_A の動作が終了したら，分割トランザクションの前半が終了したことを示す番号 ID をキューに入れる．ID は

複数の分割トランザクションが存在する場合に，どの前半が終了したのかを見分けるために各分割に対して一意に与えられる．一方，状態機械 M_B はキューの先頭に ID が現れるまでは動作を開始できない．キューの先頭に ID が現れると M_B はこれをキューから取り除いて動作を開始してもよい．このようにして，キューを用いることで，分割トランザクションを動作モデルに反映する．このようにプロセスキューは分割トランザクションを意味づけるためのものであるので，モデル上ではキューの深さは無限である．しかし，実際のモニタでは有限でなければ生成不能であるので，生成時にはパラメータとして指定する．この場合，キューが一杯の場合は分割トランザクションを開始することができない．

また，キューの動作は仕様もしくはバスプロトコルがどのように分割トランザクションを処理するかによる．$A_1, A_2, \ldots, A_n$ という順番で前部分が実行されたならば必ず，$B_1, B_2, \ldots, B_n$ という順番で後部分を処理しなければならないイン・オーダの制約があれば，キューは FIFO(First-In First-Out) になる．順番には無関心なアウト・オブ・オーダの仕様であれば，キューの要素はどれから取り出してもよい．

2.3 動作モデルの決定化

以上で構築された動作モデルは基本的に非決定的な遷移を含む有限状態機械である．本論文ではハードウェア記述言語でモニタ回路を出力するので，非決定的な遷移は記述することが不可能である．そこでモデル M を決定化する．本論文ではワン・ホット・コーディングを用い，S の要素である各状態 q_i に対してブール変数 s_i を一つ導入し，q_i に遷移すれば s_i を 1 にする．つまり，q_i への全ての遷移を $(q_{j_1}, c_{j_1}, q_i), (q_{j_2}, c_{j_2}, q_i), \ldots, (q_{j_n}, c_{j_n}, q_i)$ [ただし，$q_{j_1}, q_{j_2}, \ldots, q_{j_k} \in S$] とすると，$s_i$ の遷移関数は，$s_i' = \bigvee_{k=1..n} (s_{j_k} \wedge c_{j_k})$ となる．

3. 形式検証のためのモニタ生成

3.1 入力制約

形式検証では動作のあらゆる可能性を網羅的に調べる．モジュール単体の検証において入力端子はドントケア，つまり 0 が入力されても 1 が入力されてもよいということになる．しかし，インターフェース仕様では入力される値は特定のパターンのみに限られている．つまり，何も制約を与えないで検証した場合は，想定しない入力に対しても正しく動作するかどうかを調べることになる．これでは多くの場合，モジュール自体は正しく設計されていても，モニタはエラーという結果を返す．検証対象のモジュールに正しく入力を与えるのは，周辺回路であり，検証対象となるモジュール自身ではない．検証によって本来発見すべきエラーは「想定された正しい入力に対し，仕様通りの出力がなされていない」というものである．この入力制約は別途与える必要がある．

3.2 無視フラグの導入による入力制約の反映

本論文では，モデル中に特別な状態を設けることによって入力制約を反映させる．この手法は別途入力制約の回路を設ける必要がなく，したがって検証の際のコストの増加を抑えることができる．入力制約は仕様で規定された入力系列のみを有効にするので，同様の機構をモデルに持たせる．すなわち，仕様で規定された入力系列に対する検証結果のみを正しい結果とする．仕様にない入力系列に対しての結果は考慮しない．具体的には，無視フラグ s_{Ignr} というブール変数を設け，仕様に記述されていない入力があった場合には無視フラグを立てる ($s_{Ignr} = 1$)．次節で説明するエラー状態に遷移していても，無視フラグが立っている場合は，そのエラーは仕様にない入力系列に対するもので真のエラーとはみなさない．

エラー状態でない任意の状態において，s_k から遷移可能な条件を $c_{s_k,1}, c_{s_k,2}, \ldots$ とする．条件 $c_{s_k,j}$ は動作モデルにおいて，ユーザ定義変数のベクトル値とインターフェース信号のベクトル値の組の集合を表すものであった．このうち入力の属性を持つインターフェース信号のベクトル値に対応する条件式を i とする．したがって，$c_{s_k,1}, c_{s_k,2}, \ldots$ から入力信号線に関して条件を取り出したものを $i_{s_k,1}, i_{s_k,2}, \ldots$ とする．すると，無視フラグ s_{Ignr} の次状態 における値 s'_{Ignr} を決定する式は以下のようになる．

$$s'_{Ignr} = s_{Ignr} \vee \neg(\bigvee_k s_k \wedge (i_{s_k,1} \vee i_{s_k,2} \vee \ldots))$$

3.3 エラー状態とモニタのエラー信号

モニタは検証対象となるモジュールの入出力信号を取り込み，動作が仕様を満たすかどうかを監視する．そして，仕様にない動作をした場合，違反動作を知らせるためのエラー信号を出力する．これまで説明したモデルは仕様に書かれた動作を表した有限状態機械であり，エラーを検出するようにはできていない．そのため，動作モデルにエラー状態を付加する．モデルにおいて仕様にない動作が起こるということは，各状態において遷移が指定されていない場合に相当する．この場合を明示的に扱うため，エラー状態を設ける．

エラー状態でない任意の状態において，s_k から遷移可能な条件を $c_{s_k,1}, c_{s_k,2}, \ldots$ とすると，エラー状態 s_{Err} の次状態における値 s'_{Err} を決定する式は以下のようになる．

$$s'_{Err} = s_{Err} \vee \neg(\bigvee_k s_k \wedge (c_{s_k,1} \vee c_{s_k,2} \vee \ldots))$$

この式から，s'_{Err} は s_{Err} が現在 1 であるか，エラー状態以外のどの状態においても遷移先がない場合のみ 1 となる．エラーと無視フラグを合わせて考えた場合，s_k において，i_{s_k} が成り立たない場合は「エラー状態には遷移するが，無視フラグが立っているためエラーとはみなさない動作」，c_{s_k} のいずれかが成り立つ場合は「仕様にある動作」，c_{s_k} が成り立たずかつ i_{s_k} が成り立つ場合は「真のエラー動作」と区別していることになる．したがって，エラー信号はエラー状態で無視フラグが立っていない場合のみ 1 となり，それ以外では 0 となる出力信号線とする．

3.4 無視フラグの動作例

図 5 に状態機械の例を挙げる．また例での状態遷移モデル中の各記号 i, a, b, c, x, y, z は正規表現によるインターフェース仕様上では信号線の値の組で定義されている．ここでは，SYNC(入力), ACK(出力), SEL[1:0](入力), PERMIT(出

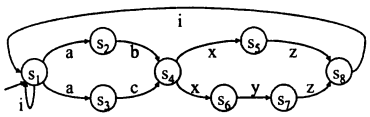

図 5　状態機械例

表 1　記号定義例

記号名	SYNC (INPUT)	ACK (OUTPUT)	SEL[1:0] (INPUT)	PERMIT (OUTPUT)
i	0	0	-	0
a	1	0	0-	0
b	1	1	00	1
c	1	1	01	1
x	1	0	1-	0
y	1	1	10	1
z	1	1	11	1

力) という信号線がモジュールにあり，各記号が表 1 のように定義されているとする．

ここで，以下の 3 種類の動作系列を仮定し，これら 3 種類の系列がそれぞれ検証対象となるモジュールのインターフェースに表れ，モニタに入力された際のモニタの動作を考えてみる．系列は (SYNC, ACK, SEL[1:0], PERMIT) のベクトル値の並びとして与える．

- 仕様通りの動作をする系列 *Correct*
- 真のエラー動作をする系列 *Error*
- 無視フラグにより検証では考慮されない系列 *Ignore*

$$\begin{aligned} Correct = <&(1,0,00,0),(1,1,01,1),(1,0,10,0),\\ &(1,1,10,1),(1,1,11,1)> \end{aligned}$$

$$\begin{aligned} Error = <&(1,0,00,0),(1,1,01,1),(1,0,10,0),\\ &(1,1,10,0),(1,1,11,1)> \end{aligned}$$

$$\begin{aligned} Ignore = <&(1,0,00,0),(1,1,01,1),(1,0,10,0),\\ &(0,1,11,1),(0,0,00,0)> \end{aligned}$$

以下で各系列に対するモデルの遷移動作を例示する．丸括弧内の 0, 1 の組は状態ベクトル $(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_{Err}, s_{Ignr})$ の値を表す．また，その後ろの角括弧はどの変数が 1 となっているかを示す．矢印は遷移を表し，その上の括弧はモニタしたインターフェース信号値の組である．

Correct の動作は以下のようになる．

$(1,0,0,0,0,0,0,0,0,0)$ $[s_1]$ $\xrightarrow{(1,0,00,0)}$
$(0,1,1,0,0,0,0,0,0,0)$ $[s_2,\ s_3]$ $\xrightarrow{(1,1,01,1)}$
$(0,0,0,1,0,0,0,0,0,0)$ $[s_4]$ $\xrightarrow{(1,0,10,0)}$
$(0,0,0,0,1,1,0,0,0,0)$ $[s_5,\ s_6]$ $\xrightarrow{(1,1,10,1)}$
$(0,0,0,0,0,0,1,0,0,0)$ $[s_7]$ $\xrightarrow{(1,1,11,1)}$
$(0,0,0,0,0,0,0,1,0,0)$ $[s_8]$

Error の動作は以下のようになる．ここでは状態 [$s5$, $s6$] において，入力である SYNC と SEL が正しい値にも関わらず，出力 PERMIT が正しくないために無視フラグは立たずにエラー状態に遷移する．

$(1,0,0,0,0,0,0,0,0,0)$ $[s_1]$ $\xrightarrow{(1,0,00,0)}$
$(0,1,1,0,0,0,0,0,0,0)$ $[s_2,\ s_3]$ $\xrightarrow{(1,1,01,1)}$
$(0,0,0,1,0,0,0,0,0,0)$ $[s_4]$ $\xrightarrow{(1,0,10,0)}$
$(0,0,0,0,1,1,0,0,0,0)$ $[s_5,\ s_6]$ $\xrightarrow{(1,1,10,0)}$
$(0,0,0,0,0,0,0,0,1,0)$ $[s_{Err}]$ $\xrightarrow{(1,1,11,1)}$
$(0,0,0,0,0,0,0,0,1,0)$ $[s_{Err}]$

Ignore の動作は以下のようになる．ここでは状態 [$s5$, $s6$] において，仕様では入力である (SYNC, SEL)=(0, 11) は規定されていない．そのため無視フラグが立つ．*Ignore* のような系列は，形式検証で全ての動作系列を調べる場合には無視される．

$(1,0,0,0,0,0,0,0,0,0)$ $[s_1]$ $\xrightarrow{(1,0,00,0)}$
$(0,1,1,0,0,0,0,0,0,0)$ $[s_2,\ s_3]$ $\xrightarrow{(1,1,01,1)}$
$(0,0,0,1,0,0,0,0,0,0)$ $[s_4]$ $\xrightarrow{(1,0,10,0)}$
$(0,0,0,0,1,1,0,0,0,0)$ $[s_5,\ s_6]$ $\xrightarrow{(0,1,11,1)}$
$(0,0,0,0,0,0,0,0,1,1)$ $[s_{Err}, s_{Ignr}]$ $\xrightarrow{(1,1,11,1)}$
$(0,0,0,0,0,0,0,0,1,1)$ $[s_{Err}, s_{Ignr}]$

4. 検証実験

以上で説明したモニタ生成手法を C プログラムで実装し，検証実験を行った．生成されたモニタは Verilog-HDL で書かれており，検証対象のモジュールに接続して検証を行う．検証に用いた計算機環境は，CPU:Pentium 3.2GHz，メインメモリ:2GB,, OS:Redhat7.3. 形式検証ツールは Cadence 社の BlackTie を用いた．BlackTie は特定のサイクルまでの限定モデルチェッキングが可能である．限定モデルチェッキングでは，特定サイクルまでの全ての動作について，モニタのエラー信号が 1 にならないかどうかを調べる．実験では 80 サイクルを指定している．

検証対象は ARM の標準的なオンチップ・バスプロトコル AMBA AHB のマスタとスレーブである．AMBA AHB の回路は沖ネットワーク LSI より提供していただいたものを使用しており，Verilog-HDL で記述されている．この AHB システムは，3 つの AHB マスタ・3 つの AHB スレーブ・1 つの AHB アービタとその他の周辺回路からなる AHB システムの実設計例である．マスタは 3 つとも同じインスタンスであるが，スレーブはそれぞれ異なる．各モジュールの設計行数と内容を表 2 に示す．

実験では AMBA AHB のマスタ，スレーブのインタフェース仕様を CWL で記述した．これは AMBA AHB の各転送モードに対して基本データ転送，バースト転送，ロック転送，スプ

表 2　各モジュールの行数と内容

モジュール	設計行数	内容
マスタ	637	4Byte×32 個のバッファ
スレーブ 1	206	16KB のメモリ 4 個
スレーブ 2	556	1KB のメモリ 12 個
スレーブ 3	425	1KB のメモリ 4 個

リット／リトライ転送を含み，AMBA AHB で実際に利用されると考えられる仕様についてはカバーしている．またアドレスインクリメンタル転送においては，アドレスの値のチェックをユーザ定義変数を用いて行っている．行数はマスタが 205 行，スレーブが 286 行である．この CWL 記述から，実装したプログラムでモニタを自動生成すると，マスタ，スレーブとも約 250 のブール変数を含むモニタが生成される．実験では動作モデルのキューの深さを指定する必要があるが，今回はキューの深さを 1 として実験を行っている．これは AMBA AHB バスプロトコルにおいては，マスタからのリクエストがスレーブによって保留された場合，マスタは別のデータ転送を開始することはできず，スレーブから再開の許可が出るのを待つ．そのため，各マスタに対して同時に分割され保留される最大の転送数は 1 であるためである．

4.1　実験 1 基本動作の検証

AHB マスタおよびスレーブに，CWL 記述から自動生成したモニタを付加してモニタがエラーを検出するかどうか検証を行った．その結果，マスタは最大使用メモリ 130MB ／検証時間 54 秒で，スレーブ 2 および 3 では 150〜250MB ／ 20〜40 秒で検証可能であった．スレーブの検証により多くのメモリが必要なのは，スレーブはマスタからのリクエストを必要があれば保留するためのメモリセルを含んでいることが原因と考えられる．スレーブ 1 については，1.2GB ／ 105 秒かかったが，スレーブ 1 は多数のメモリセルを内部に含む．メモリセルは形式検証における状態変数を増やすため，検証のコストが増大する．メモリセルを取り除いた検証では 102MB ／ 15 秒と検証コストが大幅に低くなった．各場合においてモニタはエラーを検出しなかった．

4.2　実験 2 エラーの検出

AHB マスタモジュールについて，仕様とは異なる動作を意図的に挿入することでエラーを検出できるかどうかを実験した．想定したエラーは，「バースト時に HTRANS が NONSEQ から始まらない」「HGRANT を常時反転させる」「HBURST をインクリメンタル転送指定したにもかかわらず，アドレスをインクリメンタルに増加させない」の 3 つである．

モニタ回路は 3 つすべてについてエラーを検出した．いずれも場合も約 50 秒で検証を終了し，最大使用メモリ量は 120〜130MB だった．

5. むすび

本論文では，入力制約を考慮したハードウェアモジュールインターフェースの形式検証のモニタを自動生成する手法を提案した．仕様は正規表現ベースの言語 CWL を用いており，アサーションベース検証と比べてより包括的な検証に適用しやすい．また，仕様から自動的に回路を生成するため，大規模なインターフェース仕様に対してモニタを手動で設計する際のように，モニタ生成過程で誤りが混入することがない．さらに，モニタにはモジュール単体の形式検証の際に必要な入力制約が反映されており，別途入力制約を付加する必要もない．実験では標準的なオンチップ・バスプロトコルである AMBA AHB からマスタとスレーブのインターフェース仕様を CWL で記述し，実際にモニタ回路を生成させ実験を行ったところ，実用的な時間で検証することができた．今後はより効率の良いモニタ生成手法を検討し，さらに大規模なシステムの形式検証においてもモニタベース検証を適用することが課題である．

謝辞　本稿は一部 (株) 半導体理工学研究センター (STARC) との共同研究による成果を含む．当時共同研究において，ご指導およびご討論頂いた STARC 上級研究員の小澤時典氏，客員研究員の鈴木敬氏，岩下洋哲氏，湯井丈晴氏，浅野哲也氏，多田敏彦氏に深謝したい．

文　献

[1] Kanna Shimizu, David L. Dill, Alan J. Hu, "Monitor-Based Formal Specification of PCI" *Proceedings of the Third International Conference of Formal Methods in Computer-Aided Design*, 2000.

[2] Abhik Roychoudhury, Tulika Mitra, SR Karri, "Using formal techniques to Debug the AMBA System-on-Chip Bus Protocol" *Proceedings of Design Automation and Test in Europe*, 2003.

[3] Hue-Min Lin, Chia-Chih Yen, Che-Hua Shih, Jing-Yang Jou, "On Compliance Test of On-Chip Bus for SOC" *Proceedings of the Asia and South Pacific Design Automation Conference*, pp.328-333 2004.

[4] Koji Ara, Kei Suzuki, "A Proposal for Transaction-Level Verification with Component Wrapper Language" *Proceedings of Designer's Forum of Design Automation and Test in Europe*, 2003.

[5] ARM Ltd., "AMBA Specification Rev 2.0", 1999.

[6] Hitachi Ltd., Fujitsu Ltd., Fujitsu Laboratories Ltd., "Component Wrapper Language Specification Rev 1.1", 2002.