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

2008－SLDM－135（3）
2008／5／8

局所変数を含むアサーションに対するモデルチェッキングのためのチェッカ生成

竹内 翔† 浜口 清治† 垣内 洋介† 柏原 敏伸†

† 大阪大学大学院情報科学研究科 〒560-8531 大阪府豊中市待兼山町 1-3
E-mail: †{hama,kakiuti,kashi}@ist.osaka-u.ac.jp

あらまし 機能検証のための手段としてアサーションを用いたモデルチェッキングが注目されている．SVA では，アサーション中に局所変数を用いることができるが，モデルチェッキングを考えた場合，局所変数の値を保存するための保存変数が一般に多数必要となり，計算量の点で問題となる．本稿では，モデルチェッキングに必要なメモリ量を削減することを目標に，保存変数の個数を各局所変数に対して 1 個に抑えるようなチェッカ生成アルゴリズムを提案する．提案アルゴリズムと Long と Seawright による先行研究との比較実験を行い，遅延回路や FIFO 回路に対して検証時のメモリ使用量が 10〜30% 改善されることを確認した．

キーワード モデルチェッキング，アサーション，局所変数，SystemVerilog

Checker Generation of Assertions with Local Variables for Model Checking

Sho TAKEUCHI†, Kiyoharu HAMAGUCHI†, Yosuke KAKIUCHI†, and Toshinobu KASHIWABARA†

† Graduate School of Information Science & Technology, Osaka University Toyonaka, Osaka, 560-8531, JAPAN
E-mail: †{hama,kakiuti,kashi}@ist.osaka-u.ac.jp

Abstract To perform functional formal verification, model checking for assertions has been used. It is difficult, however, to handle assertions with local variables such as in SystemVerilog. The problem is that many storing variables for local variables are required, and having many storing variables increases the computational complexity of model checking. In this report, we show an algorithm for verification in order to reduce the number of storing variables. In particular, our algorithm requires only one storing variable for each local variable. We also show experimental results for our algorithm compared with the previous work by Long and Seawright, in which the memory requirement decreases by 10-30%.

Key words Model Checking, Assertion, Local Variable, SystemVerilog

1. まえがき

近年，ハードウェアに対する効率的な検証手法の 1 つとして，アサーションベース検証が注目されている．この手法は，ハードウェア設計が満たすべき性質をアサーションとして記述し，設計中で成立しているかどうかをシミュレーションあるいは形式的手法によって検証するアプローチである．標準アサーション記述言語としては SystemVerilog Assertion（SVA）[1] や Property Specification Language（PSL）[2] などがある．

局所変数は設計記述には出現せずアサーション記述中にのみ出現する変数であり，主にデータを一時的に保存するために用いられる．たとえば，FIFO システムにおいて入力データと出力データの整合性をチェックするために使用される．SVA は局所変数を含むアサーション記述言語であるが，このようなアサーションをモデルチェッキングにより検証することは計算量の点で困難である．アサーション記述中に出現する局所変数が 1 個のみであるとしても，検証アルゴリズムにおいては局所変数に代入された値を保存するための変数（以下では保存変数と呼ぶ）が一般に多数必要になるためである．モデルチェッキング [3], [4] は形式的検証手法の 1 つであり，設計とアサーション

を入力として，設計の全ての動作に対してアサーションが成立するかどうかを網羅的に検証する手法であるが，状態変数の個数が増加すると計算量が急激に増大するため，保存変数が多数必要となることは問題である．

モデルチェッキングを行うためには，与えられたアサーションをチェッカと呼ばれる有限状態機械へと変換し，そのチェッカの遷移関数を生成しなければならない．この手続きは「アサーションのエンコード」と呼ばれている．Long と Seawright が示したアルゴリズムでは，SVA のサブセットを対象にして，局所変数を含むアサーションの長さに比例する数の保存変数が必要となる．これに対して，本稿では，同様のサブセットを対象にして，保存変数の個数を 1 つの局所変数に対して 1 個に抑えてチェッカを生成するアルゴリズムを示す．

遅延回路と FIFO 回路を対象とした実験では，文献 [7] によって得られるチェッカを使った場合に比べ，メモリ使用量が 10%〜30%改善されることが確認できた．

以下，2 章では関連研究に述べ，3 章では本稿で扱うアサーションの構文と意味論を説明する．4 章ではアルゴリズムを，5 章では実験結果を示す．

2. 関連研究

局所変数を含まないアサーションのエンコードに関しては，SVA [8]，PSL [9] それぞれについてアルゴリズムが示されている．SVA の局所変数を含むアサーションに対するモデルチェッキング問題は，一般に EXPSPACE 完全となることが示されている [10]．チェッカの生成に関しては，Long と Seawright によるアルゴリズム [7] がある．このアルゴリズムでは，局所変数への代入が含意演算子の左辺にのみ出現するようにアサーションに制限を課しており，具体的には左辺の "or" 演算子による分岐と "[*1:$]" 演算子によるループ分岐に対してそれぞれ 1 個の新しい論理変数を導入し，局所変数に対応す保存変数をパイプライン状に接続してチェッカを生成する．各局所変数に対する保存変数の個数は，アサーションのサイズに関して線形となる．

筆者らは [12] において，1 つの局所変数に対する保存変数の個数を 1 つに抑えたチェッカ生成法を示しているが，これは含意演算子の左辺に 1 つの局所変数に対して 1 回のみ代入が出現する場合のみを扱ったものであり，また限定モデルチェッキングを前提としていた．本稿の提案アルゴリズムは，文献 [7] と同様の SVA のサブセットについて，1 つの局所変数に対する保存変数の個数を 1 個に抑えたチェッカを生成することができる．

3. アサーション

3.1 構　　文

本稿で対象とするアサーションの構文 S を図 1 のように定義する．A はアサーション、P はプロパティ、R はシーケンスと呼ばれる．ここで，b を論理式、v を局所変数名，e を式とする．

この構文 S は SVA のアブストラクト構文 [1] のサブセットである．SVA のアブストラクト構文は SVA の基本的な構文を示したもので，全ての SVA 記述はアブストラクト構文を用いて記述することができる．たとえば，サイクル非

A	::=	**always assert property** P	
P	::=	R	// シーケンス
	|	(P)	// プロパティ
	|	$(R$ |-> $R)$	// 含意
R	::=	b	// 論理式
	|	$(1, v = e)$	// 局所変数への代入
	|	(R)	
	|	R ##1 R	// 連結 (オーバラップなし)
	|	R ##0 R	// 連接 (オーバラップあり)
	|	R **or** R	// 選択
	|	R [* 0]	// 0 回の繰り返し (空系列)
	|	R [* 1:$]	// 1 回以上任意回の繰り返し

図 1　本稿で扱うアサーションの構文 S

重複の含意演算 "R |=> R" はアブストラクト構文を用いて "(R ##1 1) |-> R" として，また 0 回以上の繰り返し演算 "R [*0 : $]" は "$R$ [*0] **or** R [*1 : $]" として記述可能である．

構文 S はアブストラクト構文のサブセットであるので，一部の SVA 記述は記述できない．たとえば，構文 S のシーケンスレベルでは 2 つのシーケンスの直積を意味する **intersect** 演算が定義されていないため，**intersect** 演算を用いて定義される **within** 演算と **throughout** 演算は直接的には記述できない．ただし，**intersect** 演算を除けば，サイクル非重複含意演算や 0 回以上繰り返し演算を含むほとんどの演算は構文 S を用いて記述することができる．本稿での定義では，含意演算子がネストした記述，例えば，R_1 |->R_2 |->P は許していないが，このような記述は単一の含意演算子を使って，R_1##0R_2 |->P のように書き換えることができる．

以下では含意演算子（"|->"）の左側のシーケンスを LHS (Left-Hand Side)，右側のシーケンスを RHS (Right-Hand Side) と呼ぶ．含意演算子が存在しない場合は RHS とする．

3.2 意　味　論

設計は有限状態機械であるので，ある状態において各論理変数 b_i にはそれぞれ値 0 か値 1 のいずれかが割り当てられる．各論理変数 b_i への割り当ての全ての組み合わせの集合をアルファベット Σ と定義する．Σ の各文字 (各要素) に対して論理変数への割り当てが決まっていることになるため，各文字のもとでは時間に関する演算子を含まない論理式の真偽を判定できることができる．Σ 上の文字を並べたものを語と呼ぶ．

まず，構文 S の記述に関して，有限語 u が与えられたもとでの各記述の「成立」について説明する (図 2)．ここでは，x, y, z は有限語を表す．ただし，語 u の最初の文字は 0 番目の文字であるとし，$u^{i..j}$ $(i \leq j)$ は語 u の i 番目から j 番目までの全ての文字を並べた語とする．また，$u^{i..}$ $(i \geq 0)$ は語 u の i 番目以降の全ての文字を並べた語とする．

SVA の意味論は文献 [1] の付録 E に示されている．本稿で扱うアサーションは SVA のサブセットであるので，意味論も文献 [1] に従っている．簡単のために，ここでは非決定性オートマトンの形式で説明する．アサーション P は拡張された正規表現であるとみなすことができるので，局所変数への代入と参照を除けばアサーションを非決定性オートマトンに変換できる．

- "R_1 |-> R_2" $\Longleftrightarrow$ R_1 が成立するような全ての語 $u^{0\cdots j}$ に対して，語 $u^{j\cdots}$ のもとで R_2 が成立する．
- "b" $\Longleftrightarrow$ 語 u の最初の文字のもとで論理式 b が真と評価される．
- "R_1 ##1 R_2" $\Longleftrightarrow$ 語 x に対して R_1 が成立し，かつ語 y に対して R_2 が成立し，かつ $u = xy$ となるような語 x, y が存在する．
- "R_1 ##0 R_2" $\Longleftrightarrow$ 語 xy のもとで R_1 が成立し，かつ語 yz のもとで R_2 が成立し，かつ $u = xyz$ となり，かつ語 y の長さが 1 となるような語 x, y, z が存在する．
- "R_1 or R_2" $\Longleftrightarrow$ 語 u のもとで R_1 が成立するか，あるいは語 u のもとで R_2 が成立する．
- "R_1[*0]" $\Longleftrightarrow$ 語 u の長さが 0 である．
- "R_1[*1 : $]" $\Longleftrightarrow$ $u = u_1u_2\cdots u_i$ となり，かつ全ての $u_1, u_2, \cdots u_i$ のもとで R_1 が成立するような整数 i $(i \geq 1)$ が存在する．

図 2　構文 S の記述の意味

以下では，まず局所変数が存在しない場合について考える．

ここでは，与えられた設計上の有限語を u とし，u に対してアサーション P が失敗するかどうかをチェックことを考える．まず，P に対して非決定性オートマトン M_P を生成する．このオートマトンは受理状態集合 A と失敗状態集合 F を持っている．ここで失敗状態は，各状態において遷移先が明示されていない入力が与えられた場合の遷移先である．

解くべき問題は u の全てのサフィクスが M_P に受理されるかどうかをチェックすることと等価である．SVA ではアサーションは全てのサイクルからはじまる系列をチェックすることになっているため，全てのサフィクスを考慮する必要がある．ここで，語 $u = u'u''$ としたとき，u' を u のプレフィクス，u'' を u のサフィクスと呼ぶ．

M_P に対する入力系列として u のサフィクス w を考える．このとき，M_P 中の複数の状態遷移系列すなわち「スレッド」を考えなければならない．全てのサフィクス w に対してあるスレッドが受理状態に到達するときアサーション P は u に対して「成立する」といい，あるサフィクス w に対して全てのスレッドが失敗状態に到達するときアサーション P は u に対して「失敗する」という．これら以外の場合は「検証途中である」という．以上が [1] の付録 E に示されている意味論の非決定性オートマトンを用いた解釈となる．

本稿では，アサーションが失敗するかどうかをチェックすることに焦点を当てる．つまり，「失敗」と「検証途中または成立」を区別する．

含意演算子（|->）の扱いには特に注意が必要である．ここでは，アサーション P |=> Q を考える (簡略化のため，構文 S で定義されている |-> の代わりに，|=> を用いて説明する)．P が u に対して失敗しないことをチェックするためには，u の全てのサフィクス w に対して以下をチェックしなければならない．ここで，$w = w'w''$ である．

> M_P の受理状態に到達するスレッドが存在する全ての w' に対して，M_Q の失敗状態以外の状態に到達するスレッドが w'' に対して存在する．

すなわち，上記のような全ての w' に対して，M_Q が w'' に対して失敗しないということを調べなければならない．

次に，局所変数への代入と参照が存在する場合を考える．本稿では，局所変数への代入を LHS でのみ許している．局所変数に関係する SVA の意味論においては，各スレッドで独立して代入を扱う必要がある．M_P が入力系列 w' に対して 2 つのスレッド t_1 と t_2 を持っているとする．局所変数 v への代入が t_1 に対しては 2 サイクル目で t_2 に対しては 3 サイクル目で発生するとき，M_Q を評価する際に v に保存されている値は一般に異なる．これら 2 つのスレッドは独立に扱わなければならない．

このような意味論はチェッカ生成時に問題になる．部分集合構成法のような標準的な手法で M_P を決定化すると 2 つのスレッドを併合してしまう．その結果，上記の例では 2 サイクル目の代入が 3 サイクル目の代入で上書きされてしまい，正しい結果を得ることができなくなる．

3.3　アサーションの制限

対象とするアサーションは以下の制限を全て満たすものとする．

（1）　局所変数への代入は LHS でのみ記述できる．

（2）　記述 "R_1 |-> R_2" 中 R_2 が R[*0] であってはならない．

（3）　記述 "R_1 ##0 R_2" 中 R_1，R_2 のいずれかが R[*0] であってはならない．

1. については，たとえば多くのアサーション記述例を集めている [11] などでも，局所変数への代入がインプリケーション演算子の右辺で用いられている例はなく，大きな支障はないと考えられる (文献 [7] でも同じ制限が仮定されている)．また，2. と 3. のいずれかに該当するようなアサーションは，意味論上どのような系列に対しても常に失敗するため実際には扱う必要がない．

4.　アルゴリズム

本章では，与えられた設計とアサーションに対して，チェッカを生成するアルゴリズムとモデルチェッキングを行う際に与えるべきプロパティを示す．モデルチェッキングを行うためには，チェッカは決定的に動作するような記述になっている必要がある．

アルゴリズムは次の 3 ステップからなる．

（1）　アサーションを非決定性オートマトンに変換する．

（2）　オートマトンから決定性オートマトンに対応する遷移関数を生成する．

（3）　モデルチェッキングのためのプロパティを生成する．

4.1　アサーションの変換

与えられたアサーションを非決定性オートマトンへ変換する．アサーションは一種の正規表現とみなすことができることから，標準的な手法により再帰的に変換することができる．3.2 節で述べたように，ここでは，失敗状態集合を導入している．どのような系列に対しても，あるスレッドが失敗状態以外の状態に到達すれば，アサーションは失敗しないと判断される．たとえ

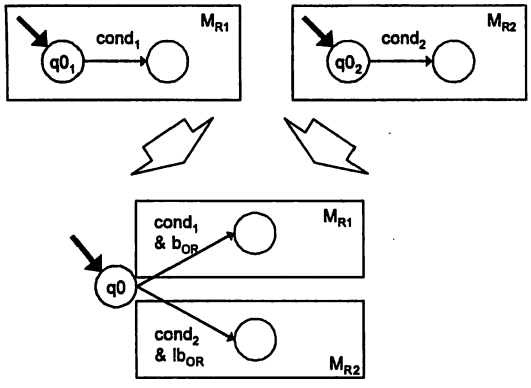

図 3 R_1 or R_2

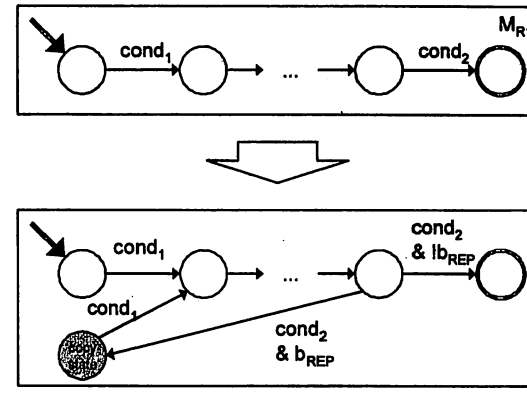

図 4 R_1 [* 1:$]

ば，SVA のシークエンス表現 ab に対して，オートマトンは ab のみを失敗でないと判定するだけでなく a も失敗でないと判定する．これ以外に，SVA の非明示的な "always" を扱う必要があるが，これについては，後の遷移関数の生成の際に考慮する．以下では，変数への代入，含意演算子の左辺での非決定的分岐の処理について述べる．含意演算子の左辺での非決定性分岐については，このステップで決定化する．

変数 v への代入を意味する "$(1, v = e)$" に対しては，次の項目を組として記録しておき遷移関数を生成する際に用いる．

- 局所変数名 v
- 代入される値を表す式 e
- 代入がおこる遷移 (現状態，次状態，遷移条件の組).

次に含意演算子の左辺に対応する非決定性オートマトンの決定化について述べる．3.2 節で述べたように，通常の部分集合構成手法では正しく局所変数を扱うことができない．決定化は非決定的な分岐に対して新たに論理変数を導入することによって行う．以下では，"**or**" 演算子と "[*1 : $]" 演算子に対する処理を示す．なお，(局所変数を含まない) 右辺に関しては非決定性オートマトンのまま残しておき，遷移関数を生成する際にまとめて決定化を行う．以下，$trans(i, j, e)$ がある遷移に対して真であるとは，その遷移の遷移元が i，遷移先が j，遷移条件が e であることを意味するものとする．

記述 'R_1 **or** R_2" に対しては，まず，オートマトン R_1 と R_2 に対応するオートマトン M_{R_1} and M_{R_2} をそれぞれ生成する．M_{R_1} と M_{R_2} のいずれか一方を決定的に選択するために，新しい論理変数 b_{OR} を導入する．q_0 を M_{R_1} の初期状態とする．$trans(q_0, s, cond)$ が真であるような遷移それぞれについて，遷移条件 $cond$ を新しい遷移条件 "$cond \wedge b_{OR}$" で置き換える (図 3 参照)．M_{R_2} に対しては，遷移条件 $cond$ を "$cond \wedge \neg b_{OR}$" で置き換える．これにより，$b_{OR} = 1$ の場合には，M_{R_1} が $b_{OR} = 0$ の場合には M_{R_2} がそれぞれ選択されることになる．

記述 "R_1 [* 1:$]" に対しては，まずオートマトン M_{R_1} を構成し，初期状態に対して新たに複製状態を導入する．次に，R_1 の繰り返しに対応する遷移を付加する．これは，M_{R_1} の受理状態へ 1 遷移のみで到達する状態から，複製状態への遷移を付け加えることによって行う．「R_{R_1} の繰り返し」および「R_{R_1} からの離脱」を決定的に選択するために，新しい論理変数 b_{REP} を導入する．S を M_{R_1} の受理状態へ 1 遷移のみで到達する状態の集合とする．$\exists s \in S$, $trans(s, s', cond)$ が真であるような M_{R_1} の各遷移に対して，s' が受理状態集合に含まれているときには，$cond$ を $cond \wedge \neg b_{REP}$ に変更し，s' が複製状態のときは，$cond$ を $cond \wedge b_{REP}$ に変更する (図 4 参照)．これにより，$b_{REP} = 1$ の場合には，R_1 の繰り返しが選択され，$b_{OR} = 0$ の場合には繰り返し部分を抜け出すことになる．

上記以外でも，空系列に対応する "$R_1[*0]$" があれば非決定性の分岐が発生する．この場合についても新しい論理変数を導入することで，同様に決定化を行うことができる．

次状態のない遷移はモデルチェッキングで許容されないため，受理状態と失敗状態に対して true をラベルした自己ループの遷移を加える．以下では，失敗状態はすべて併合して 1 つであると仮定している．

4.2 遷移関数の生成

次に前節で得られた非決定性オートマトンから遷移関数を生成する．同時に決定化も行う．

まず，開始サイクルを選択できるように非決定性オートマトンを修正する．アサーションはすべてのサイクルでチェックを始めることになっているため，オートマトンもすべてのサイクルでスタートできるようにしなければならない．ここでは，新しく 2 つの論理変数 b_{START} と b_{CHECK} を導入する．これら 2 つの変数に対する遷移関数を以下に示す．$next(b)$ は b の次のサイクルでの値を意味し，記法 $\{a_0, \cdots, a_n\}$ は，$a_0, \cdots, a_n$ から 1 つの値が非決定的に選ばれることを意味する．b_{START} の初期値は 0 または 1．b_{CHECK} の初期値は 0 である．オートマトンは $b_{START} = 1$ のときのみ動作を開始し，それ以外の場合は，初期状態で待ち続ける．

$$next(b_{START}) = \begin{cases} 0 & (b_{START} \vee b_{CHECK}) \\ \{0,1\} & (\text{それ以外}) \end{cases}$$

$$next(b_{CHECK}) = \begin{cases} 1 & (b_{START} \vee b_{CHECK}) \\ 0 & (\text{それ以外}) \end{cases}$$

次に，決定化と遷移関数の生成を行う．ここではいわゆるワンホットコーディングを用いる．Q をオートマトンの状態集合とする．$s \in Q$ に対して，新しい論理変数 b_s を導入する．以下にそれぞれの論理変数 b_s に対する遷移関数を示す．ここで，$S_{prev}(s)$ は s から 1 遷移で到達可能な状態の集合を表す．$cond(s, s')$ は，s から s' へ遷移にラベルされている遷移条件である．s が初期状態なら b_s の初期値は 1 であり，そうでなければ 0 である．

$$next(b_s) = \bigvee_{i \in S_{prev}(s)} (b_i \wedge cond(i, s))$$

変数 b_{OR}, b_{REP} に関しては，常に 0 または 1 のいずれかを非決定的に選択するとする．初期値も 0 または 1 のいずれかである．

$$next(b_{OR}) = \{0,1\}, \quad next(b_{REP}) = \{0,1\}$$

各局所変数 v に対して，保存変数 st_v を導入する．それぞれの代入に関しては，アサーションをオートマトンに変換する段階で，局所変数名 v と代入する値を表す式 e，および遷移 (遷移元 s ，遷移先 s'，遷移条件 $cond$) が記録されている．$(v, (s, s', cond), e)$ によってこれを表現することとする．v に対して $lv_i = (v_i, (s_i, s_i', cond_i), e_i)$ $(i = 1, \cdots, n)$ であるとする．遷移関数は次のようになる．ここで，含意演算子の左辺はすでに決定化されていることから，2つ以上の条件が同時に真になることはない．

$$next(st_v) = \begin{cases} e_0 & (b_{s_0} \wedge cond_0 \text{ is true}) \\ \vdots & \\ e_n & (b_{s_n} \wedge cond_n \text{ is true}) \\ st_v & (\text{otherwise}) \end{cases}$$

以上の遷移関数すべての論理積がチェッカの記述となる．

4.3 プロパティの生成

前節の遷移関数を設計と結合したのち，モデルチェッキングを行う．モデルチェッキングには次のプロパティを用いる．このプロパティは線形時間時相論理 (Linear-Time Temporal Logic, LTL) で記述されている．$\mathbf{G}p$ は p が常に成り立つことを意味している．Q, Q_{LHS} , q_F は，それぞれ状態集合，含意演算子の左辺に対応するオートマトンの状態集合，失敗状態を表している (失敗状態は 1 つに併合されている)．このプロパティが満足されなければ，与えられたアサーションは失敗 (不成立) であるということになる．

$$\mathbf{G} \neg (b_{q_F} \wedge \bigwedge_{s \in (Q - Q_{LHS} - \{q_F\})} (\neg b_s))$$

最終的に得られる遷移関数の記述は新たな論理変数として $b_{START}, b_{OR}, b_{REP}$ を含んでいる．b_{START} がいつ 1 になるかによって，アサーションのチェックが開始されるサイクルが決まり，また，各サイクルでの b_{OR} および b_{REP} の値によって，含意演算子の左辺に出現する非決定的分岐についてどの分岐を選ぶかが決まる．$b_{START}, b_{OR}, b_{REP}$ の値が各サイクルで特定の値に固定された場合を考えると，含意演算子の右辺に対応する決定性オートマトンによって，与えられた入力系列 (設計の動作系列) が失敗かどうかが判定されることになる．モデルチェキングのアルゴリズムは，これらの論理変数が各サイクルでとり得る値すべてについて，また，すべての設計の動作系列について，プロパティのチェックを行うため，結果としてすべての場合をカバーしたモデルチェッキングが達成される．

5. 実　　験

アルゴリズムを C 言語で実装し，文献 [7] の手法との比較実験を行った．CPU : Pentinum 4 2.4GHz, メモリ : 2GB, OS : Linux 2.4.20-8 の実験環境で，モデルチェッカ NuSMV を用いて限定モデルチェッキングを行っている．

遅延回路と FIFO 回路の 2 つの設計に対して，提案手法と文献 [7] の手法の双方で検証を行い，検証に要した時間とメモリ使用量を計測した．

遅延回路の入力信号は reqin, データ線については入力として din, 出力として dout を持つ．遅延は固定値 5 である．動作は，1) reqin=1 が成立したとき，データ din を格納し，2) 格納したデータを 5 サイクル後に dout に出力するというものである．アサーションは遅延回路の入力データと出力データの間のデータ整合性をチェックするものである．x は局所変数である．

```
always assert property (
  (reqin, x=din) |-> (##5 dout==x)
)
```

このアサーションを構文 S に従った形式に書き直すと以下のようになる．

```
always assert property (
  (reqin ##0 (1, x=din))
     |-> (1 ##1 1 ##1 1 ##1 1 ##1 1 ##1 dout==x)
)
```

FIFO 回路の入力は reqin と reqout であり，各 4 ビットの内部信号 incnt, outcnt を持つ．データ線は入力として din, 出力として dout を持つ．FIFO の深さは固定値 10 である．動作は : 1) reqin=1 が成立したとき，データ din とタグ incnt を格納し，incnt をインクリメントする．2) reqout=1 が成立したとき，その 1 サイクル後に格納したデータを dout に出力し，outcnt をインクリメントする．アサーションは FIFO 回路の入力データと出力データの間のデータ整合性をチェックするものである．ここで，x, tag は局所変数である．

```
always assert property (
  ((reqin, x=din, tag=incnt) ##[1:$]
                  (reqout && tag==outcnt))
      |-> (##1 d_out==x)
)
```

このアサーションを構文 S に従った形式に書き直すと以下のようになる．

```
always assert property (
  ((reqin ##0 (1, x=din) ##0 (1, tag=incnt))
    ##1 1[*1:$] ##1 (reqout && tag==outcnt))
 |-> (1 ##1 d_out==x)
)
```

表 1 遅延回路に対する検証時間 (秒)

k	$width$	提案手法	[7]	改善率
10	4	0.6	0.7	14 %
	6	3.5	4.2	17 %
	8	35.2	38.3	9 %
15	4	2.0	2.2	10 %
	6	16.6	16.7	1 %
	8	188	188	0 %

表 2 FIFO 回路に対する検証時間 (秒)

k	$width$	提案手法	[7]	改善率
10	4	23.7	29.1	19 %
	6	184	211	13 %
	8	1022	1215	16 %
15	4	144	258	46 %
	6	804	1101	27 %
	8	8167	10308	21 %

表 3 保存変数の全ビット数

k	$width$	提案手法	[7]	改善率
10	4	4	20	80 %
	6	6	30	80 %
	8	8	40	80 %
15	4	8	32	75 %
	6	10	40	75 %
	8	12	48	75 %

限定モデルチェッキングの限界長：10，15，データのビット幅 $width$：4，6，8 ビットの場合について，検証時間と検証に要したメモリ量の最大値を計測した．提案手法による検証時間と文献 [7] の手法による検証時間を，遅延回路と FIFO 回路について示す (表 1 および表 2)．遅延回路，FIFO 回路共にデータのビット幅が増加すると，検証時間の改善率は悪化することがわかる．これは，データのビット幅が増加することで，検証時間を増加させる主要な要因がアサーション部分から設計部分へと変化しているためであると考えられる．検証時間をどの程度改善できるかは検証対象の設計及びにアサーションに大きく依存しており，本手法の効果は限定的なものになると考えられる．

保存変数に必要なビット数に関して，提案手法と文献 [7] の手法とを比較した結果を表 3 に示す．保存変数のビット数は文献 [7] の手法に比べて 75%〜80%減少している．

遅延回路と FIFO 回路に対する必要記憶量を示す (表 4 および表 5)．遅延回路，FIFO 回路共に 10%〜30%程度改善していることがわかる．アサーション部分の局所変数のための保存変数の個数が提案手法と文献 [7] の手法で大きく異なることが原因と考えられる．

6. むすび

本稿では，局所変数を含む SVA アサーションに対するチェッカ生成アルゴリズムを示した．各局所変数に対して，必要とされる保存変数の数が 1 である点が文献 [7] と異なる特徴である．FIFO 回路，遅延回路について実装・実験行い，必要メモリ量

表 4 遅延回路に対する必要記憶量 (MB)

k	$width$	提案手法	[7]	改善率
10	4	11.6	13.1	12 %
	6	19.7	25.9	24 %
	8	52.7	77.0	32 %
15	4	13.1	15.2	14 %
	6	25.6	34.1	25 %
	8	77.9	114	32 %

表 5 FIFO 回路に対する必要記憶量 (MB)

k	$width$	提案手法	[7]	改善率
10	4	26.3	31.1	16 %
	6	70.2	83.0	16 %
	8	252	293	14 %
15	4	40.2	62.1	36 %
	6	117	160	27 %
	8	450	600	25 %

が 10-30% 削減されることを示した．今後の課題としては，扱うことのできるシンタクスの拡張の他，モデルチェッキング向きのより効率のよい状態割当て法の考案があげられる．

文　献

[1] IEEE Std 1800-2005 - "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language", IEEE Computer Society, 2005.

[2] Property Specification Language (PSL). http://www.mel.nist.gov/psl/

[3] E. M. Clarke, Jr., Orna Grumberg, and Doron A. Peled: "Model Checking", The MIT Press, 1999.

[4] K. L. McMillan: "Symbolic Model Checking", Kluwer Academic Publishers, 1993.

[5] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu: "Symbolic Model Checking Using SAT Procedures instead of BDDs", Design Automation Conference, pp.317–320, 1999.

[6] A. Cimatti, E. Clarke, F. Giunchi, and M. Roveri: "NuSMV: A New Symbolic Model Verifier", International Conference on Computer-Aided Verification, 1999.

[7] Jiang Long and Andrew Seawright: "Synthesizing SVA Local Variables for Formal Verification", Design Automation Conference, pp.75-80, 2007.

[8] Sayantan Das, Rizi Monhanty, Pallab Dasgupta, and P.P. Chakrabarti: "Synthesis of SystemVerilog Assertions", Design Automation and Test in Europe, pp.70-75, 2006.

[9] Katell Morin-Allory, and Dominique Borrione: "Proven Correct Monitors from PSL Specifications", Design Automation and Test in Europe, pp.1246-1251, 2006.

[10] Doron Bustan and John Havlicek: "Some Complexity Results for System Verilog Assertions", Computer-Aided Verification, LNCS 4144, pp.205–218, 2007.

[11] Harry D. Foster, Adam C. Krolnik and David J. Lacey: "Assertion-Based Design" 2nd Edition, Kluwer Academic Publishers, 2004.

[12] S. Takeuchi, K. Hamaguchi, T. Kashiwabara: "Encoding Assertions with Dynamic Local Variables for Bounded Property Checking", Synthesis and Simulation Meeting and International Interchange, pp.515-521, 2007.