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

2006－SLDM－127（3）
2006／11／28

動的局所変数を含むアサーションに対する限定モデルチェッキング

竹内 翔† 浜口 清治† 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: †{s-takeut,hama,kashi}@ist.osaka-u.ac.jp

あらまし 回路をフォーマルに検証する手法として，アサーションに対する限定モデルチェッキングが注目されている．しかしながら，SystemVerilog に見られるような動的局所変数を含んだアサーションは計算量の点で扱いが困難である．そこで，本稿では動的局所変数を含むアサーションに対する限定モデルチェッキングについて主として必要メモリ量を削減するためのアルゴリズムを検討する．アルゴリズムを実装して実験した結果も示す．
キーワード アサーションベース検証，限定モデルチェッキング，動的局所変数，SystemVerilog

Bounded Model Checking for Assertions including Dynamic Local Variables

Sho TAKEUCHI†, Kiyoharu HAMAGUCHI†, and Tosinobu KASHIWABARA†

† Graduate School of Information Science and Technology, Osaka University
Machikaneyama-cho 1-3, Toyonaka-shi, 560-8351 Japan
E-mail: †{s-takeut,hama,kashi}@ist.osaka-u.ac.jp

Abstract To perform functional formal verification, bounded model checking for assertions has been proposed. For bounded model checking, however, it is difficult to handle assertions including dynamic local variables such as in SystemVerilog. In this report, we investigate an algorithm for verifying assertions including dynamic local variables with bounded model checking. This algorithm focuses on reduction in memory requirement. We implemented the algorithm and performed some experiments.
Key words Assertion-Based Verification, Bounded Model Checking, Dynamic Local Variable, SystemVerilog

1. はじめに

近年，半導体回路が大規模化，複雑化するにともない，回路の検証を効率的に行うことがより一層重要視されるようになってきている．回路の効率的な検証手法として，アサーションベース検証が提案されている．アサーションベース検証とは，回路設計が満たすべき仕様をブール式とそれらの時間的な関係を表す式からなるアサーションとして記述して，そのアサーションが真と評価されることを動的（シミュレーション）または静的（形式的）に検証する手法である．アサーション記述言語には，SystemVerilog [1] や PSL [2] などがある．

形式的検証とは，数学的な証明によって設計の正しさを網羅的に調べる手法である．形式的検証には，ある回路と別の回路の機能が等価であることを検証する等価性検査と，ある設計に対して仕様が満たされていることを検証するモデルチェッキングがある．本稿では，後者のモデルチェッキング，特にアサーションを用いたモデルチェッキングに注目する．

アサーションを用いたモデルチェッキングでは，設計を状態遷移機械としてモデル化し，アサーションが偽と評価されるような状態系列が存在するかどうかを網羅的に検証する [3], [4]．しかしながら，網羅的な検証を行うために，状態数爆発が問題となる．状態数爆発問題を緩和するために有効な手法に限定モデルチェッキングがある．

限定モデルチェッキングは，初期状態から高々k サイクル（k は自然数，外部から与える）までの間でアサーションが偽と評価されるような状態系列が存在するかどうかを検査する．このとき，k の値を適切に定めることで限定的ではあるが網羅的な検証が可能となる [5]．

以上のように，アサーションを用いた限定モデルチェッキングは有用であるが，アサーション記述によっては検証が困難な場合も存在する．とくに，SystemVerilog における動的局所変数を含むアサーションは計算量の点で扱いが難しい．動的局所

```
ASSERT_PROPERTY_STATEMENT ::=
 "assert" "property" "("
   {ASSERTION_VARIABLE_DECLARATION} PROPERTY_SPEC
 ")"
PROPERTY_SPEC ::=
 PROPERTY_EXPR
PROPERTY_EXPR ::=
   SEQUENCE_EXPR
 | SEQUENCE_EXPR "|->" SEQUENCE_EXPR
 | SEQUENCE_EXPR "|=>" SEQUENCE_EXPR
SEQUNCE_EXPR ::=
   EXPR
 | EXPR BOOLEAN_ABBREV
 | CYCLE_DELAY_RANGE SEQUENCE_EXPR
 | SEQUENCE_EXPR CYCLE_DELAY_RANGE SEQUENCE_EXPR
 | "(" EXPR "," { DYNAMIC_VARIABLE_ASSIGNMENT } ")"
CYCLE_DELAY_RANGE ::=
   "##" m
 | "##" "[" m ":" n "]"
BOOLEAN_ABBREV ::=
   "[" "*" m "]"
 | "[" "*" m ":" n "]"
DYNAMIC_VARIABLE_ASSIGNMENT ::=
   VARIABLE_IDENTIFIER "=" EXPR
```

図 1 構 文 S

変数とは後に参照するためにデータを一時的に保存しておくための変数のことである．動的局所変数はたとえば，FIFO システムで入力されるデータと出力されるデータが一致していることを検証するために利用できる．問題は，1 つの動的局所変数を用いて記述されたアサーションを検証する場合でも，検証ではデータを保存するための変数を 1 つ用意しただけでは十分でないという点である．たとえば，前述の FIFO システムの場合では，検証で用いるデータ保存用の変数が 1 個では，新しくデータが入力された場合に以前のデータを上書きしてしまい，正しく検証が行えない．一般に，アサーション内で定義された動的局所変数の個数よりも多数の変数が検証時には必要となる．本論文ではこのような動的局所変数を含むアサーションを限定モデルチェッキングで検証するためのアルゴリズムについて検討する．

以下，2 章では本稿で用いるアサーションと限定モデルチェッキングを説明し，3 章では提案アルゴリズムの説明を行い，4 章では実験結果を示し，5 章ではまとめと今後の課題を述べる．

2. 準 備

2.1 対象とするアサーション記述

本稿で対象とするアサーションの構文 S は図 1 のとおりである．構文 S は SystemVerilog の構文のサブセットである．

アサーション記述の意味を以下で簡単に説明する．SystemVerilog のアサーションでは，以下のような記述を与えたとき，それが毎サイクル評価されることを要請している．

- P1 ##[m:n] P2

サイクル遅延を表す．P1 が終了した m 以上 n 以下サイクル後に P2 が開始されることを意味する．

- P1 [*m:n]

繰り返しを表す．P1 を m 以上 n 以下の回数だけ繰り返すことを意味する．

- P1 |-> P2

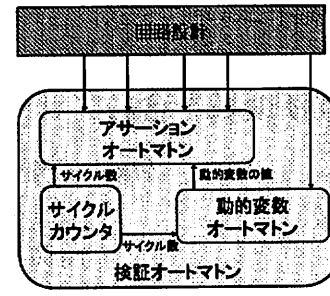


図 2 全 体 図

P1 を前提条件，P2 を帰結とした含意を意味する．ただし，P1 の最後のサイクルと P2 の最初のサイクルが時間的に同時に発生する．

- P1 |=> P2

P1 を前提条件，P2 を帰結とした含意を意味する．ただし，P1 が終了した次のサイクルに P2 が開始される．

[アサーションに対する制約]

動的局所変数への値の代入はアサーション中のシーケンス記述の最初の 1 サイクル目のみに出現するものとする． □

この制約を満たしたアサーションの例を以下に示す．

```
assert property (
  int x;
  (insig , x = d_in) |=> ##4 d_out==x
)
```

このアサーションは，“信号 insig が 1 になったときに入力データ d_in を動的局所変数 x に保存し，その 4 サイクル後に出力データ d_out と x が等しくなる”ことを要求している．動的局所変数への値の代入はシーケンス記述の最初の 1 サイクル目にのみ出現しているので，これは制約を満たしている．

3. アルゴリズム

本章では，回路設計（有限状態機械）とその設計に対するアサーションが与えられたときに，アサーションが偽と評価される状態遷移系列が存在するかどうかを限定モデルチェッキングを用いて検証する手法を示す．具体的には，図 2 のように検証オートマトンを作成し，回路設計と結合して検証を行う．

3.1 検証オートマトンの作成方法

図 2 のように検証オートマトンはアサーションオートマトン，動的変数オートマトン，サイクルカウンタから構成される．各部分の作成方法を以下で示す．

3.1.1 アサーションオートマトンの作成

まず，与えられたアサーションを以下の条件を満たす非決定性オートマトンに変換する．

- オートマトンの状態間の遷移はアサーション記述の1サイクルに対応する.
- アサーションが真と評価される状態遷移系列が存在する場合にのみ, 受理状態に到達できる.
- その状態からは以後どのように遷移をしてもアサーションが真と評価される状態遷移系列にはならないことを意味する特別な状態 stateF が存在する.
- 受理状態, 状態 stateF はそれぞれ自己ループを持つ.

このオートマトンの作成はアサーションの構文 S を解析しながら行う. 具体的な変換アルゴリズム convert は以下のようになる (図3).

[アルゴリズム convert]

(入力) アサーション文 P

(出力) 非決定性オートマトン

(方法) P の形に応じて以下のように再帰的に変換する. ここでは, 特徴的な構文のみをとりあげる.

(1) P が " 論理式 expr " の場合

状態集合を{s0, s1, stateF}として, 遷移関数 δ が δ(s0, expr) = s1, δ(s0, ¬expr) = stateF となり, 初期状態が s0 で, 受理状態が s1 であるようなオートマトンを作成して, このオートマトンを返す.

(2) P が " 論理式 expr , 代入文 assign " の場合

P が " 論理式 expr " の場合と同様のオートマトンを返す. 代入文 assign に関しては, 論理式 expr と代入先変数名と代入する式を記録しておく. これは後述の動的変数オートマトンを作成するために必要な情報である.

(3) P が " P1 [* m : n] " の場合

まず, convert(P1) を再帰的に呼び出し, P1 に対応するオートマトン $\mathcal{A}_{P1}$ を作成する. 次に, $\mathcal{A}_{P1}$ を n 個コピーしてそれらを順に ϵ 遷移で接続する. そして, (n - m) 個目までの全てのコピーに対して, 自身の初期状態から自身の受理状態への ϵ 遷移を追加する. 最後に, 全体のオートマトン $\mathcal{A}_P$ の初期状態と受理状態をそれぞれ 1 個目のコピーの初期状態と n 個目のコピーの受理状態として, 全ての ϵ 遷移を除去した等価なオートマトンへと変換して返す.

(4) P が " P1 ## [m : n] P2 " の場合

まず, convert(P1), convert(P2) を再帰的に呼び出し, P1, P2 に対応するオートマトン $\mathcal{A}_{P1}$, $\mathcal{A}_{P2}$ をそれぞれ作成する. 次に, m サイクル以上 n サイクル以下の遅延を表すオートマトン $\mathcal{A}_{delay}$ を作成する. そして, $\mathcal{A}_{P1}$ から $\mathcal{A}_{delay}$ への ϵ 遷移と, $\mathcal{A}_{delay}$ の受理状態から $\mathcal{A}_{P2}$ の初期状態への ϵ 遷移を追加する. 最後に, 全体のオートマトン $\mathcal{A}_P$ の初期状態と受理状態をそれぞれ $\mathcal{A}_{P1}$ の初期状態と $\mathcal{A}_{P2}$ の受理状態として, 全ての ϵ 遷移を除去した等価なオートマトンへと変換して返す.

(5) P が " P1 |-> P2 " の場合

まず, convert(P1), convert(P2) を再帰的に呼び出し, P1, P2 に対応するオートマトン $\mathcal{A}_{P1}$, $\mathcal{A}_{P2}$ をそれぞれ作成する. このとき, もし $\mathcal{A}_{P1}$ に stateF が含まれている場合には, その stateF の状態名を変更した後に $\mathcal{A}_{P1}$ の受理状態集合に含める. 次に,

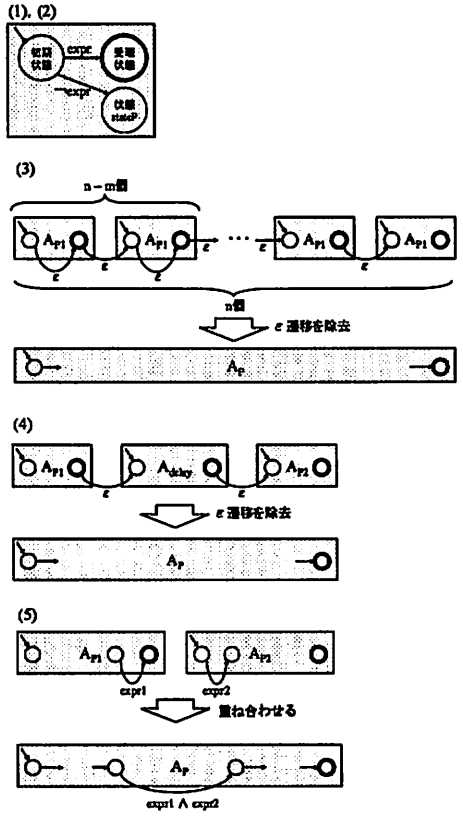


図 3 アルゴリズム convert

$\mathcal{A}_{P1}$ の最後の遷移と $\mathcal{A}_{P2}$ の最初の遷移を重ねるために以下のような処理を行う. $\mathcal{A}_{P1}$ の遷移先状態が受理状態となる全ての遷移 (δ(状態 $s1$, 論理式 $expr1$) = 受理状態) と, $\mathcal{A}_{P2}$ の遷移前状態が初期状態となる全ての遷移 (δ(初期状態, 論理式 $expr2$) = 状態 $s2$) に対して, 新しい遷移 ($\delta(s1, expr1 \wedge expr2) = s2$) を追加して, $\mathcal{A}_{P2}$ の遷移前状態が初期状態となる全ての遷移を削除する. 最後に, 全体のオートマトン $\mathcal{A}_P$ の初期状態を $\mathcal{A}_{P1}$ の初期状態とし, 受理状態を $\mathcal{A}_{P2}$ の受理状態とした後に返す.

さらに, アルゴリズム convert で作成した非決定性オートマトンの全ての受理状態と状態 stateF に自己ループを作成する.

3.1.2 検証オートマトンとその決定化

サイクル数をカウントするサイクルカウンタ *counter* を以下のように定める. ここで, ' 付きの変数は次状態変数を表す. *counter* の初期値は 0 である.

$$counter' = counter + 1$$

各動的局所変数を保存するための変数を考える. ここで, アサーション制約から, 動的局所変数への代入が起こるのはアサーション中のシーケンス記述の 1 サイクル目のみであるので, 動的局所変数保存用変数は動的局所変数 1 個あたり, 各サイクルに 1 個必要となる. しかしながら, このままでは $k+1$ 個 (k は限定モデルチェッキングの限界長) の動的局所変数保存用変

数が必要となり，メモリ使用量が大きくなると予想される．そこで，オートマトンをいくつかに分離することを考える．具体的には，動的局所変数を n サイクル分 $(1 \leq n \leq k+1)$ だけ用意して，検証オートマトンを $\lceil (k+1)/n \rceil$ 個作成し，別々に検証を行うという手法を考える．そのために，まず，各検証オートマトン中で，何サイクル目の動的局所変数を保存しているかを示す次のような変数 $index$ を導入する．

$$index' = index$$

この変数 $index$ の初期値を ni から $n(i+1)-1$ $(i = 0, 1, \cdots, \lceil (k+1)/n \rceil\text{-}1)$ の範囲で非決定的に選択するように制約をおくことで，その検証オートマトンが ni サイクル目から $n(i+1)-1$ サイクル目までに対応するようにする．

ここで，アサーションオートマトンを決定化して，各遷移がサイクルカウンタ $counter$ の値が $index$ 以上の場合のみに起こるように変更する．アサーションオートマトンの初期状態以外の各状態 i に対応するブール変数 b_i と初期状態に対応するブール変数 b_{init} を導入してオートマトンを決定化する．状態 i に到達したときに $b_i = 1$ となり，そうでない場合は $b_i = 0$ となる．各ブール変数 b_i と b_{init} の遷移は以下のようになる．b_i の初期値は 0，b_{init} の初期値は 1 である．ただし，状態 i を遷移後の状態とする全ての遷移の集合を $TRset_i$ とする．また，引数で指定した遷移における遷移前の状態を返す関数を getcurrent，引数で指定した遷移における遷移条件式を返す関数を getcond とする．

$$b'_{init} = counter < index$$

$$b'_i = (counter \geq index) \wedge \vee_{tr \in TRset_i} (b_{\mathrm{getcurrent}(tr)} \wedge \mathrm{getcond}(\mathrm{tr}))$$

次に，1 個の検証オートマトン中の動的局所変数保存用変数の遷移を考える．前述のように，1 個の動的局所変数 x に対して，その保存用変数は n 個必要となる．ここで，検証オートマトンが ni サイクル目から $n(i+1)-1$ サイクル目までに対応していると仮定して，保存用変数を x_j $(ni \leq j \leq n(i+1)-1)$ と書くことにする．動的局所変数への代入が発生するような遷移と代入後の式は既にアルゴリズム convert で求まっているので，そのような遷移を tr とし，代入後の式を new とすると，変数 x_j の遷移は以下のようになる．これを動的変数オートマトンと呼ぶ．

$$\begin{aligned} &((counter \geq index) \wedge (j = index) \\ &\quad \wedge b_{\mathrm{getcurrent}(tr)} \wedge \mathrm{getcond}(\mathrm{tr}) \wedge x'_j = new) \\ \vee\ &((\neg(counter \geq index) \vee \neg(j = index) \\ &\quad \vee \neg b_{\mathrm{getcurrent}(tr)} \vee \neg\mathrm{getcond}(\mathrm{tr})) \wedge x'_j = x_j) \end{aligned}$$

この式は，サイクルカウンタ $counter$ の値が $index$ 以上でかつ動的局所変数への代入が発生した場合には $index$ に対応する動的局所変数保存用変数を更新し，そうでなければ以前の値を保持するということを表している．

以上より，変数 $counter$，$index$，アサーションオートマトンを表す各ブール変数 b，動的局所変数保存用変数 x_j $(ni \leq j \leq n(i+1)-1)$ の式を全て $\wedge$ 演算子で接続すると，ni サイクル目から $n(i+1)-1$ サイクル目までに対応する検証オートマトンが作成できる．このとき，$index$ の初期値は ni から $n(i+1)-1$ までの範囲で非決定的に選択する．

このような操作を繰り返して，$\lceil (k+1)/n \rceil$ 個の検証オートマトンを作成して，それぞれの検証オートマトンを別個に検証する．各検証オートマトンの遷移関数は注目しているサイクルの範囲に関する記述が異なるのみで，それ以外の記述は同一である．

3.2 限定モデルチェッキングを用いた検証方法

限定モデルチェッキングの限界長を k，グルーピングサイズを n とすると，検証オートマトンは $\lceil (k+1)/n \rceil$ 個となる．検証を行う場合には，検証オートマトンを 1 個選択して設計回路と接続した後に，接続して作成されたオートマトンに対して以下のようなアサーションを限定モデルチェッキングを用いて検証する．実際には線形時間論理を用いて記述している．

- 各サイクルのいずれの時点でも受理状態に到達しない
- k サイクル目に状態 stateF 以外の状態に到達していない

上記の条件の両方を充足するような状態系列が存在すれば，回路設計中に元々のアサーションを満たさないような状態系列が存在する，すなわち回路設計が仕様を満たしていないということになる．このような検証を全ての検証オートマトンに対して行う．

グルーピングサイズ n の値によって検証時間とメモリ使用量は変化する．後の実験でも示すように，n の値を大きくするとメモリ使用量が大きくなる傾向があり，一方，$n = 1$ とすると検証時間が増大する傾向がある．そこで，メモリ使用量をできる限り小さくしてかつ検証時間がそれほど大きくならないような検証アルゴリズムを考える．回路設計とそれに対するアサーション，限定モデルチェッキングの限界長 k が与えられたとすると，アルゴリズムは以下のようになる．

[検証アルゴリズム]

```
n := 1;  n_pre := 1;  SetA_pre := 空;  // 初期化
// グルーピングサイズを決定するループ
while (true) {
  SetA_cur := {グルーピングサイズ n の検証オートマトン};
  SetA_cur から検証オートマトン A を 1 個選択して検証;
  time_cur := 検証時間;  mem_cur := メモリ使用量;
  SetA_cur から A を削除;
  if (mem_cur >= MAXMEM)
    SetA_cur := SetA_pre  break;
  alltime_cur := estimate(time_cur);
  rate_time := getRate(alltime_cur, alltime_pre);
  if (rate_time >= TIMERATE)
    SetA_cur := SetA_pre;  break;
  n_pre := n;
  n += CN;
  if (n > (k+1))
    SetA_cur := SetA_pre;  break;
```

```
  if (n == (k+1))  break;
  alltime_pre := alltime_cur;
  SetA_pre := SetA_cur;
}
// 実際に全ての検証を行う
SetA_cur 中の全ての検証オートマトンを検証;
return;
```

ここで，アルゴリズム中の値 $MAXMEM$，$TIMERATE$，CN と関数 estimate，getRate は以下を意味する．

- $MAXMEM$

メモリ使用量の限界値を表し，外部から与える．メモリ使用量がこの値以上になった時点でグルーピングサイズを固定する．

- $TIMERATE$

関数 getRate() で計算される時間改善率の最低値を表し，外部から与える．時間改善率がこの値以下になった時点でグルーピングサイズを固定する．

- CN

グルーピングサイズの増加量を表し，外部から与える．

- estimate()

引数に指定された 1 個の検証オートマトンの検証時間 $time$ から，全ての検証オートマトンの検証時間を推定する関数である．後の実験で示すように，各検証オートマトンの検証時間はほとんど同じである．ただし，$\lceil (k+1)/n \rceil$ 個目の検証オートマトンだけは対応しているサイクル数が他の検証オートマトンと異なる場合があるので（$k+1$ が n で割り切れない場合），この場合も考慮して全体の検証時間を以下のように推定する．$q = (k+1)/n$，$r = (k+1) \bmod n$ とすると，$time \times (q + \frac{r}{n})$ を全体の検証時間と推定して返す．

- getRate()

引数に指定された今回の全検証時間と前回の全検証時間から，時間改善率を計算する関数である．時間改善率は今回の検証が前回の検証からどれだけ早くなったかを表すもので，$\frac{(\text{前回の全検証時間} - \text{今回の全検証時間})}{\text{前回の全検証時間}}$ で計算される．時間改善率が大きいほど前回よりも検証時間が早くなっていることを意味し，負の値の場合は前回よりも遅くなっていることを意味する．

4. 実験結果

本節での実験環境は以下のとおりである．

- CPU : Intel Celeron 1200MHz
- 物理メモリ : 504.02MB
- OS : Linux 2.6.0-test5_2
- 限定モデルチェッカ : NuSMV [6]

3 章で説明したアルゴリズムを実装して実験を行った．

まず，グルーピングサイズ n とメモリ使用量との関係を調べるために，次のような遅延回路に対して実験を行った．

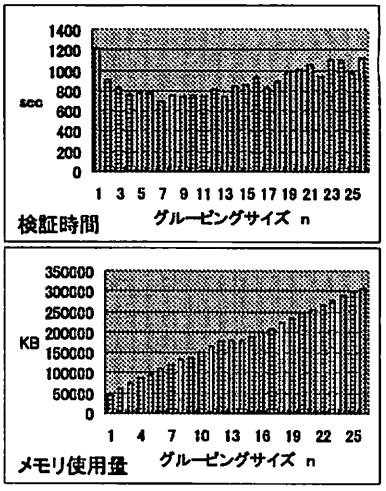


図 4　グルーピングサイズとメモリ使用量の関係

信号	:	入力 in
データ	:	入力 d_in，出力 d_out
遅延回路の長さ	:	len
		入力データを len サイクル後に出力
動作	:	in=1 のとき，d_in を取り込み，len サイクル後にそのデータを出力

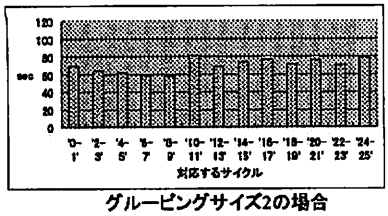
この回路に対して，データ整合性を検査する以下のようなアサーションを記述する．

```
assert property (
  int x;
  (in , x = d_in) |=> ##len d_out==x
)
```

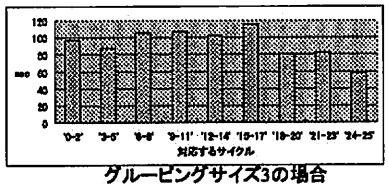
限定モデルチェッキングの限界長 $k = 25$，遅延回路の長さ $len = 5$，ビット幅 4bit として，グルーピングサイズ n を $1 \leqq n \leqq k+1 = 26$ の範囲で変化させたときの検証時間（全ての検証が終了するまでの時間）とメモリ使用量を図 4 に示す．図 4 より，グルーピングサイズが増加するとメモリ使用量が増加することがわかる．よって，メモリ使用量を小さくするには，グルーピングサイズを小さくすればよい．しかし，図 4 より，グルーピングサイズを最小の 1 にすると検証時間が大きくなってしまう．さらに，グルーピングサイズを大きくすれば，検証時間が小さくなるとは限らない．そこで，メモリ使用量をできるだけ小さくし，かつ検証時間がそれほど大きくならないようにするためには，3.2 節で説明した検証アルゴリズムを用いる．

また，同実験でグルーピングサイズが 2 の場合と 3 の場合における，各検証オートマトンの検証時間を図 5 に示す．図 5 より，各検証オートマトンの検証時間はそれほど変化していないことがわかる．よって，1 個の検証オートマトンの検証時間から，全ての検証オートマトンの検証時間を推定するには 3.2 節の検証アルゴリズムで説明した関数 estimate() を利用すればよい．

次に，3.2 節で説明した検証アルゴリズムの性能を調べるた

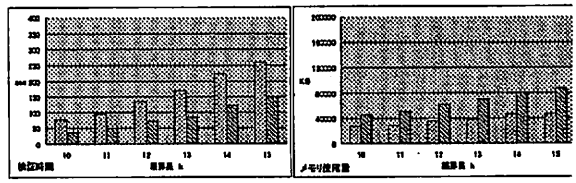


グルーピングサイズ2の場合

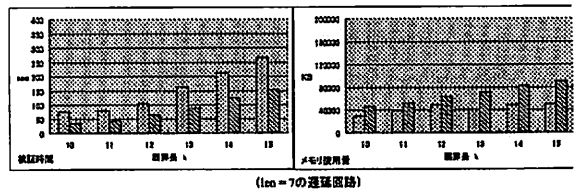


グルーピングサイズ3の場合

図 5　各検証オートマトンの検証時間

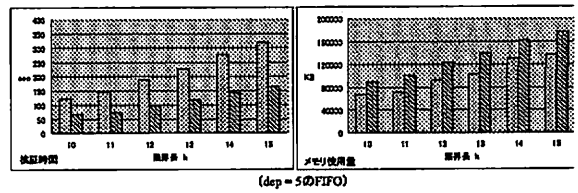


(len = 5の遅延回路)

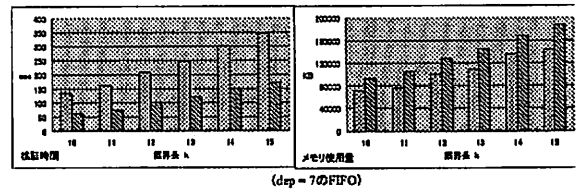


(len = 7の遅延回路)

図 6　遅延回路の実験結果（左バー：手法 A　右バー：手法 B）



(dep = 5のFIFO)



(dep = 7のFIFO)

図 7　FIFO の実験結果（左バー：手法 A　右バー：手法 B）

めに，前述の遅延回路と以下のような FIFO 回路に対して実験を行った．

信号	:	入力 in，out
データ	:	入力 d_in，出力 d_out
内部信号	:	full，empty
FIFO の深さ	:	*dep*
		FIFO に保存できる最大データ数が *dep*
動作	:	full=0 かつ in=1 のとき d_in を取り込む
		empty=0 かつ out=1 のとき d_out に出力
		FIFO 内のデータが d 個のとき，full は 1
		FIFO 内のデータ 0 個のとき，empty は 1

この回路に対するデータ整合性を検査する以下のアサーションを考える．

```
assert property (
  int x,tag;
  (in && !full , x = d_in , tag = count) |->
      ##[1:$] out ##1 d_out==x
)
```

ビット幅 4bit として *len*，*dep* が 5，7 の場合に，k を 10 から 15 まで変化させたときの，提案アルゴリズムで検証した場合（手法 A）とはじめから $n = k + 1$ として検証した場合（手法 B）との検証時間とメモリ使用量を測定した．実験結果は遅延回路に関しては図 6，FIFO に関しては図 7 のとおりである．ただし，検証アルゴリズムのパラメータについては，$MAXMEM$ = 400MB，$TIMERATE$ = 0.20，$CN = 4$ とした．

図 6，7 より，提案アルゴリズムは，アルゴリズムの繰り返し部分のオーバーヘッドによって実行時間は遅くなるものの，メモリ使用量では有利であることがわかった．

5.　まとめと今後の課題

本論文では，動的局所変数を含むアサーションに対する限定モデルチェッキングを用いた検証アルゴリズムを考案し，そのアルゴリズムを実装して実験結果を示した．実験結果より，提案アルゴリズムはグルーピングサイズ決定のためのオーバーヘッドのため実行時間は遅くなるものの，メモリ使用量を抑えることができることを示した．

今後の課題はグルーピングサイズ決定にかかるオーバーヘッドを減少させたより効率的な検証アルゴリズムの開発である．また，図 4 のような検証時間となる原因を解明することも重要な課題である．

文　献

[1] System Verilog. http://www.systemverilog.org/

[2] Process 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, Y. Zhu: "Symbolic Model Checking Using SAT Procedures instead of BDDs", Proc. 36th Conference on Design Automation, pp.142-170, vol.98-2, 1992.

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