IPSJ Transactions on System LSI Design Methodology Vol. 2 80–92 (Feb. 2009)

Regular Paper

Checker Generation of Assertions with Local Variables for Model Checking

Sho Takeuchi,[†1,*1] Kiyoharu Hamaguchi[†1] and Toshinobu Kashiwabara[†1]

To perform functional formal verification, model checking for assertions has attracted attentions. In SystemVerilog, assertions are allowed to include "local variables", which are used to store and refer to data values locally within assertions. For the purpose of model checking, a finite automaton called "checker" is generated. In the previous approach for checker generation by Long and Seawright, the checker introduces new state variables corresponding to a local variable. The number of the introduced state variables for each local variable, is linear to the size of a given assertion. In this paper, we show an algorithm for checker generation in order to reduce the number of the introduced state variables . In particular, our algorithm requires only one such variable for each local variable. We also show experimental results on bounded model checking for our algorithm compared with the previous work by Long and Seawright.

1. Introduction

Hardware systems have continued getting larger in scale and more complex, and it becomes more and more important to verify systems at their design process. For this purpose, assertion-based verification has been proposed. Assertion-based verification is a method, in which specifications are described as assertions and the assertions are checked in dynamic method (e.g., simulation) or in static method (e.g. model checking). An assertion is a certain kind of a temporal logic formula, which expresses temporal relationship between boolean formulas. There are standardized assertion description languages such as SystemVerilog Assertion (SVA) [1], Property Specification Language (PSL) [2].

Model checking is a formal verification method, which checks exhaustively whether a given assertion holds on a given design. Model checking has attracted attentions as a verification method for assertion-based verification.

In SVA, we can use local variables in assertions. Local variables are used to store and refer to data values locally within assertions and are used, for example, for checking data consistency of a FIFO system.

In this paper, we focus on model checking of assertions with local variables. In order to perform model checking, the given assertion is converted into a finite automaton called "checker". This conversion procedure is called "encoding assertions". Long and Seawright have proposed an algorithm for converting an SVA assertion into a checker [8]. In order to handle local variables, their algorithm introduces "storing variables", or "storage", to store data values in a generated checker. Here, we distinguish local variables used in an assertion from the corresponding storing variables in a generated checker. The algorithm by Long and Seawright produces a checker with storing variables of which number is linear to the size of the assertion.

We can expect that reducing the number of storing variables in a checker can improve the computational requirement in model checking. In this paper, we show a checker generation algorithm for verifying assertions using only one storing variable for each local variable. Since our algorithm does not change much the number of computational steps required in the model checking procedure, the improvement in terms of run-times is limited. In this sense, the contribution of this paper is mainly the improvement in memory requirement.

We show experimental results for comparing with the previous work by Long and Seawright. In this experiment of bounded model checking, we check data consistency for a delay circuit and a FIFO circuit and measure the verification time and memory requirement. Our algorithm improved by 10–30% on memory requirement for both the delay circuit and the FIFO circuit.

We explain the related works in Section 2, assertions we handle in this paper in Section 3, our algorithm in Section 4, and show experimental results in Section 5, respectively. Section 6 concludes this paper.

2. Related Works

In some previous works [9),10)], an encoding algorithm for assertions without

†1 Osaka University
∗1 Presently with Hitachi Government and Public Corporation System Engineering Corp

© 2009 Information Processing Society of Japan

local variables has been proposed. As a theoretical result, the model checking problem for SVA assertions with local variables has been shown as EXPSPACE-complete [11].

Long and Seawright have proposed an encoding algorithm for assertions with local variables [8]. In their work, they restrict that the substitution for local variables occurs only at the left-hand side of implication operators. In their algorithm, they introduce a new input variable for each branch originated from "**or**" operator and "[∗1:$]" operator. As is stated in Section 1, they introduce storing variables, which correspond to local variables in a given assertion. They deploy a cascade of storing variables for local variables. The number of storing variables for each local variable is linear to the size of an assertion.

Furthermore, the assertion assumes "abstract syntax" of SVA [1], where a description such as "##100" is expanded by one hundred of '##1's. This implies 100 storing variables in a checker for each local variable. As compared with their work, our algorithm requires only one storing variable for each local variable.

In a previous work [12], a checker generation method was proposed, in which the number of storing variables for a local variable is one. This method, however, can handle assertions which have only one substitution for a local variable at the left-hand side of implication operators. It also assumes only bounded model checking as a formal verification method.

3. Assertions

In this section, we explain the syntax, semantics and restriction for assertions we handle.

3.1 Syntax

We define the syntax S of assertions as in **Fig. 1**, where b is a logical formula, v is a local variable name, e is a expression. A, P and R are called assertion, property and sequence respectively. Note that, in particular, the implication operator `|->` can be used only once.

In this paper, we assume "concurrent assertion", where the assertions are required to be checked at *every* cycle.

We call the left-hand side of an implication operator "LHS", and the right-hand side "RHS". If there is not any implication operator in a sequence, then the overall sequence is "RHS".

A	::=	**assert property** P	
P	::=	R	// sequence
	|	(P)	// property
	|	$(R$ `|->` $R)$	// implication
R	::=	b	// logical formula
	|	$(1, v = e)$	// substitution for
			// local variables
	|	(R)	// parenthesis form
	|	R `##1` R	// concatenation
	|	R `##0` R	// fusion
	|	R **or** R	// "or" form
	|	R [*0]	// null repetition
	|	R [*1:$]	// unbounded repetition

Fig. 1 Syntax S of assertions.

The syntax S is a subset of an abstract syntax of SVA [1]. All SVA descriptions can be written by using the abstract syntax. For example, a cycle non-overlapping implication operation "R_1 `|=>` R_2" can be described as "(R_1 `##1` 1) `|->` R_2", and null or unbounded repetition operation "R_1 [∗0:$]" can be described as "$R_1$ [∗0] **or** R_1 [∗1:$]".

Our working example is as follows:

$$(a, n = 0)\ \#\#1\ (b, n = n + 1)[*1 : \$]\ \#\#1\ c \\ \texttt{|->}\ c\ [*1 : \$]\ \#\#1\ (d\ \&\ n\ !=\ 3) \tag{1}$$

3.1.1 Abstract syntax and Syntax S

Since syntax S is a subset of the abstract syntax, some SVA formulas cannot be described. For example, an operator "*intersect*" which expresses product for two sequences is not allowed in syntax S. Thus, operators "*within*" and "*throughout*", which are defined by the operator "*intersect*", cannot be described.

This can become a problem, when we write assertions for practical designs. It is, however, known that handling "intersect" operator causes high computational complexity in model checking. Even without local variables, model checking assertions with "intersect" operator is known to be an EXPSPACE-complete problem [11]. Development of an efficient algorithm remains a future work.

All of the other operators, which include cycle non-overlapping implication

© 2009 Information Processing Society of Japan

- "R_1 |-> R_2"
  $\Longleftrightarrow$ For any word $u^{0..j}$ such that R_1 holds, R_2 holds for the word $u^{j\cdot\cdot}$.
- "b"
  $\Longleftrightarrow$ The truth of logical formula b is determined over the first letter of the word u.
- "R_1 ##1 R_2"
  $\Longleftrightarrow$ There exist words x,y such that R_1 holds for the word x and R_2 holds for the word y and $u = xy$.
- "R_1 ##0 R_2"
  $\Longleftrightarrow$ There exist words x,y,z such that R_1 holds for the word xy and R_2 holds for the word yz and $u = xyz$ and $|y| = 1$.
- "R_1 **or** R_2"
  $\Longleftrightarrow$ R_1 holds for the word u or R_2 holds for the word u.
- "R_1 [$*$0]"
  $\Longleftrightarrow$ $|u| = 0$.
- "R_1 [$*$1:$]"
  $\Longleftrightarrow$ There exists i ($i \leq 1$) such that $u = u_1 u_2 \cdots u_i$ and R_1 holds for the all words $u_1, u_2, \cdots, u_i$.

Fig. 2 Satisfiability of each description of syntax S.

operator and null or unbounded repetition operator, can be described in syntax S.

3.2 Semantics

Assertions are defined using logical variables. Since we handle finite automata, we relate logical variables in the assertions to alphabet for the finite automata as follows: a letter l in alphabet Σ is defined as a combination of assignments for logical variables used in the assertion. For example, if logical variables a, b and c are used, then Σ is $\{000, 001, 010, \cdots, 111\}$. Then, the truth-value of a logical formula without temporal operators can be determined over each letter.

In the following, we handle only *finite* words.

At first, we explain briefly the satisfiability of each description in syntax S as shown in **Fig. 2** for a given finite word u. Here, x, y, z are finite words.

The first letter of a word w is 0th letter of w. Let $u^{i..j}$ ($i \leq j$) be the word which is the sequence of letters from ith to jth in the word u, $u^{i\cdot\cdot}$ ($i \geq 0$) be the word which is the sequence of letters from ith in the word u, $|u|$ be the length of the word u.

The rigid semantics of SVA is shown in Annex E of literature [1]. In this section, for brevity, we overview it with the terminology of a nondeterministic automaton, rather than in the style of the original semantics.

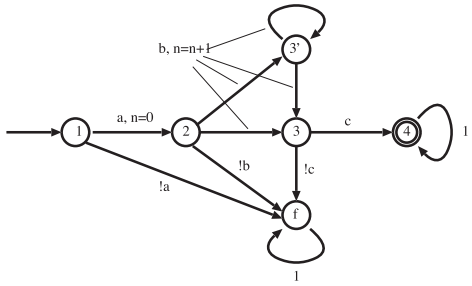

Fig. 3 A finite automaton for the LHS.

Since an assertion P can be regarded as a certain kind of regular expression, it can be converted into a nondeterministic automaton, except for substitution for local variables and reference to them.

Figure 3 shows a finite automaton corresponding to the LHS in assertion (1), where the double circle means an accepting state, and the state labeled by f is a failing state. The structure corresponding to $(b, n = n + 1)[*1 : \$]$ in the LHS may not look usual. This is introduced in this form so that we can use this automaton in the explanation of our algorithm.

Suppose that we have a finite sequence u of a given design and try to check whether assertion P does not fail for u. We can construct a nondeterministic automaton M_P for P. This automaton has an accepting state set A and a failing state set F. A failing state in F is the state to which M_P makes a transition for inputs such that no explicit transition is specified in the description of P.

The problem to solve is basically equivalent to that of checking whether none of the suffix of u fails in M_P. The reason we have to consider all the suffixes is that we assume concurrent assertions. Here, for word $u = u'u''$, we call u' "prefix" and u'' "suffix".

Furthermore, suppose that w, which is a suffix of u, is given as an input sequence for M_P, then we have multiple state transition sequences, or "threads" in M_P.

Figure 4 shows how multiple threads occur for word w, when it is given to the finite automaton in Fig. 3.

If some thread reaches an accepting state for every suffix w, then it is said that P "holds" for u. If "all" of the threads reach failing states for some suffix w, then

© 2009 Information Processing Society of Japan

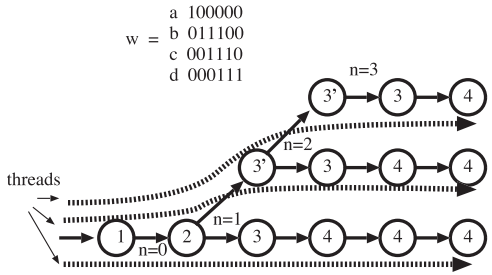

Fig. 4 Multiple threads for word w.

it is said that P "fails" for u. Otherwise, that is, if some of threads reach failing states and the rest of them reach the state which are neither accepting states nor failing states for every suffix w, then it is said that P is "pending" for u.

In this paper, we focus on checking whether an assertion fails or not, that is, we distinguish "fails" from "pending or holds". The other types of check can also be handled with some modification.

As for implication operators, we have to be careful about its semantics. Let us consider an assertion P `|->` Q. To check whether P does not fail for u, we have to check the following for every suffix w of u. Here we define $w' = w^{0..j}$ and $w'' = w^{j..}$. Note that the last letter of w' is the first letter of w''.

> For every w' which has a thread ending at an accepting state of M_P, there exists a thread for w'' ending at some state which is not in the failing state set of M_Q.

In other words, we have to check whether, for every w' accepted by M_P , M_Q does not fail for w''.

Figure 5 shows how the LHS and the RHS in assertion (1) are converted to two automata M_P and M_Q, and how they behave for a given word w. As for substitutions, we explain soon in this section. Since the LHS and the RHS of the assertion must be treated separately, the two automata are not concatenated at this step, but the edge between state 3 and state 4 in the both automata should be regarded as the same edge.

The SVA semantics related to local variables requires us to handle substitutions along *each thread* separately. Suppose that M_P has two threads t_1 and t_2 for an

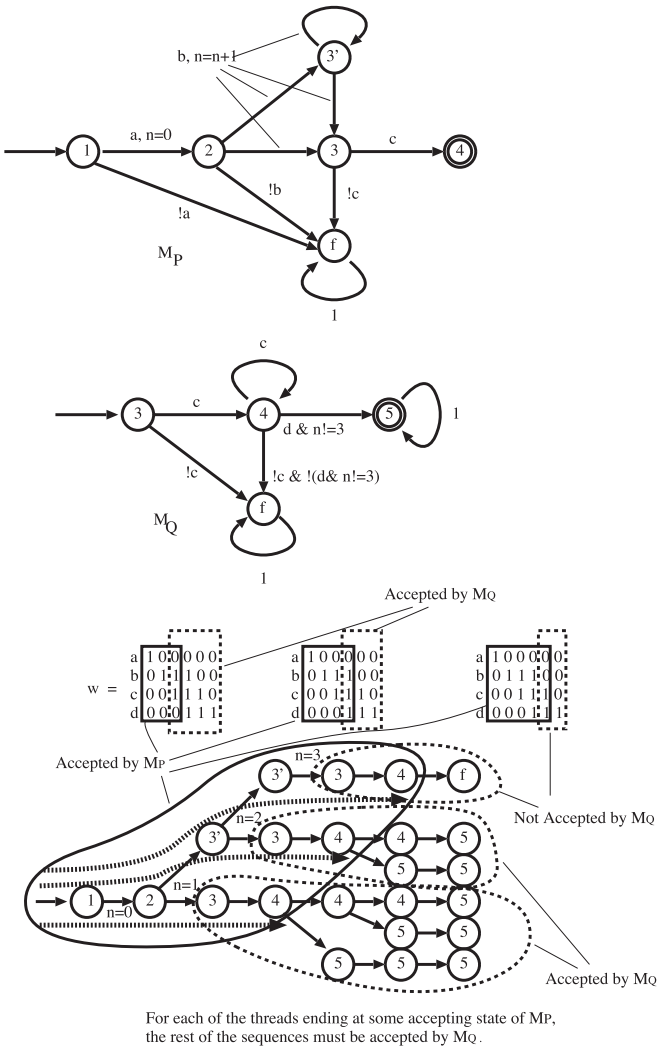

Fig. 5 Two automata for P |-> Q.

input sequence w. If t_1 has a substitution for a local variable x at the second cycle and t_2 has at the third, then the value to be stored at x can be different,

© 2009 Information Processing Society of Japan

when evaluation on M_Q starts. These two threads must be handled separately.

Figure 5 also shows how the results of substitutions in M_P should be propagated to M_Q for a given word w. This example shows substitutions along three threads.

This way of handling threads can cause a problem in generating a checker. If we determinize M_P with a standard technique such as subset construction, threads are merged and a substitution at a cycle can be overwritten by that at another cycle, which leads to an incorrect result.

3.3 Restriction

Assertions we handle must satisfy the following restriction.

(1) The substitution for local variables occurs only at LHS.
(2) In "R_1 `|->` R_2", R_2 must not be $R[*0]$.
(3) In "R_1 `##0` R_2", either R_1 or R_2 must not be $R[*0]$.

These restrictions reduce the expressive power, but we think it is rare to describe assertions which do not satisfy the restrictions. In particular, assertions which do not satisfy the restriction (2) or (3) always fail for any sequences according to the semantics. Therefore, it makes no sense to describe these assertions. Furthermore, as for restriction (1), for example, there is no such example listed in the section of "Assertion Cookbook" in one of the standard textbooks [5] on assertion-based design.

4. Algorithm

4.1 Overview

The output of our algorithm is a set of transition functions of a checker corresponding to a given assertion.

We generate a finite automaton corresponding to the assertion, and then, transition functions. Since assertions can be regarded as regular expressions, we can use a standard recursive algorithm for conversion to a nondeterministic automaton, which can be found, for example, in some standard textbook [13]. In order to perform model checking, the automaton must be determinized as in automaton-based model checking (See, for example, literature [3]). Although the determinization is done during automaton generation, this is the basic approach done in the previous works [8)–10)],

We also take the approach in this direction. The difference lies in how to handle substitutions in the LHS, and how to implement the mechanism to check the "always" property, that is, to check whether the assertion holds at every cycle. The latter is the requirement for concurrent assertions.

The algorithm is composed of 3 main steps (1)–(3) shown in the below. At each step, the following sub-steps are performed.

(1) Conversion to a finite automaton.
  (a) Conversion from an assertion to a nondeterministic finite automaton.
  (b) Storing information related to substitutions in the assertions.
  (c) Determinization of the automaton corresponding to the LHS.
  (d) Concatenation of the two automata corresponding to the LHS and the RHS.
  (e) Modification of the automaton for handling the "always" property.
(2) Generating transition functions from the automaton.
  (a) Determinization of the automaton.
  (b) Addition of substitutions to the transition functions.
(3) Generating a temporal logic formula for model checking.

Before explaining the details of the above procedures, we show the key ideas together with an example.

4.2 Key Ideas

Naturally, the automata we can obtain from assertions such as those shown in Fig. 5, can be used for checker generation. We modify them to produce a checker.

In the following, we refer to the storage to store data values in a generated checker as "storing variables". They correspond to local variables in an SVA assertion. We distinguish local variables from storing variables, depending on where the variables are used.

In order to determinize the automaton corresponding to the LHS, we introduce new logical variables, which are regarded as input variables. These variables are used to determinize the non-deterministic branches caused by "**or**" operator and "[∗1:$]" operator in the LHS. Implicitly assumed "always" for SVA concurrent assertions is handled similarly by introducing a new logical variable which selects the starting cycle of the generated checker.

Figure 6 shows how a new logical variable b_{REP} is introduced to determinize

© 2009 Information Processing Society of Japan

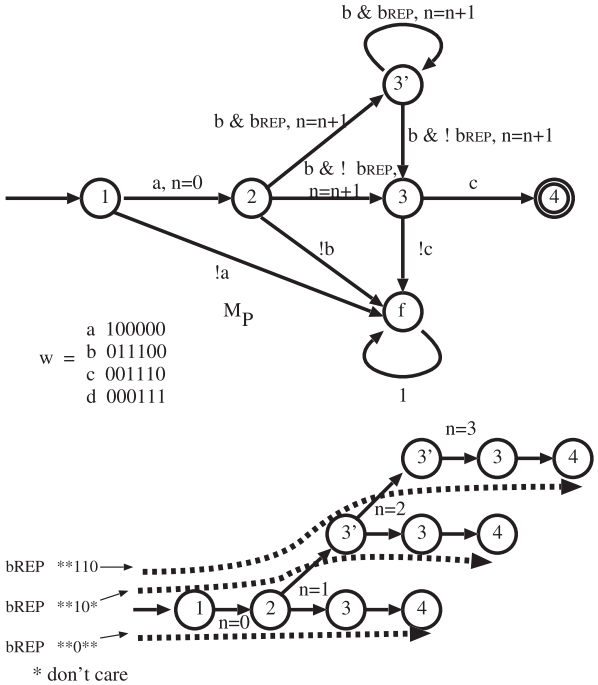

Fig. 6 Determinization by new input variables.

a nondeterministic branch in the LHS. The figure also shows how threads occur for word w, according to the combinations of the values for the logical variable b_{REP}. Note that, when values for the logical variable is fixed at each cycle, only one thread is executed. For example, if the values of b_{REP} are $**110$, then the uppermost thread runs for word w, where $*$ represents either 0 or 1 (don't care).

Figure 7 shows the result of concatenating of the two automata of the LHS and the RHS. Note that the edge from state 3 to state 4 is merged. This figure also shows how a logical variable, b_{START}, is introduced for selecting the starting cycle of the generated checker. An additional transition function forces the logical variable to become 1 only once along any possible sequence on input variables (See Section 4.4 for the detail).

In general, the new logical variables are introduced so that, once values for the

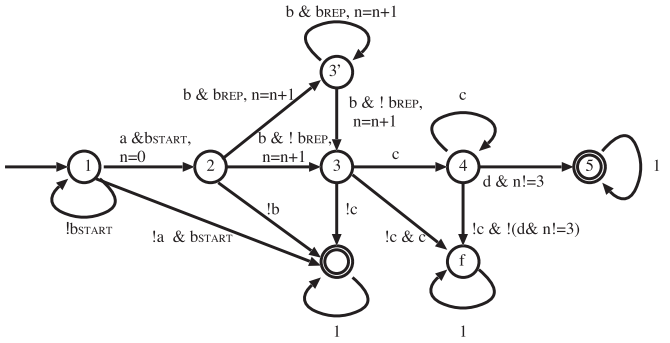

Fig. 7 A resulting automaton.

logical variables are fixed at each cycle, the checker starts only once, and only one thread of the automaton corresponding to the LHS is executed. As a result, overwriting to a local variable from different threads can be avoided. This is also the reason that only one storing variable is necessary for each local variable.

A model checker checks whether a given property holds for each of all of possible input sequences. Thus, all of possible threads starting at an arbitrary cycle are verified.

4.3 Conversion to a Finite Automaton

(a) Conversion to a nondeterministic finite automaton

We convert a given assertion into a nondeterministic automaton. Since an assertion can be viewed as a kind of a regular expression, this can be done in rather a straightforward recursive manner.

Its acceptance condition, however, is different from the conventional definition. What we are interested in is to check whether a behavioral sequence of a design does not fail the assertion. Thus, we introduce a special state called a failing state in this automaton as we explained in Section 3.2. If some of the threads reach some accepting state for any sequence (behavior) of designs, then the assertion is satisfied.

Since transitions which do not have any next state are not allowed for model checking, we add self-loop transitions labeled with 1 (true) for all the accepting states and the failing state in the resulting automaton. By constructing the

© 2009 Information Processing Society of Japan

automaton appropriately, we can have only one failing state.

(b) Storing information related to substitutions

For description "$(1, v = e)$" which indicates the substitution for a local variable, we keep records of the following items in order to generate a transition function including a storing variable at a later step.

- Local variable name v
- Substituted expression e
- Transition at which the substitution occurs, which includes a current state, a next state and a transition condition

(c) Determinization of the LHS automaton

Next, we determinize the nondeterministic automaton corresponding to LHS. The determinization can be achieved by introducing new logical variables for nondeterministic branches. We show how to handle the nondeterministic branches originated from "**or**" operator and "[∗1:$]" operator which occur in the LHS. Note that the conversion result for these operators at the RHS remains as a usual nondeterministic automaton at this step, and is determinized when transition functions are generated. Here, "$trans(i, j, e)$ is true for a transition" means that the transition has current state i, next state j and transition condition e.

For description "R_1 **or** R_2", first we make automata M_{R_1} and M_{R_2} corresponding to R_1 and R_2 respectively. In order to select M_{R_1} or M_{R_2} deterministically, we introduce a new logical variable b_{OR}. Let q_0 be an initial state of M_{R_1}. For any transition such that $trans(q_0, s, cond)$ is true for M_{R_1}, we modify the transition condition $cond$ to a new transition condition "$cond \wedge b_{OR}$" as shown in **Fig. 8**. For M_{R_2}, we modify the transition condition $cond$ to a new transition condition"$cond \wedge \neg b_{OR}$" similarly. Intuitively, if $b_{OR} = 1$ holds, then the automaton selects M_{R_1}, and if $b_{OR} = 0$ holds, then the automaton selects M_{R_2}.

For description "R_1 [∗1:$]", we at first make automaton M_{R_1} and add a copy state corresponding to an initial state. Next, we add transitions for repetition of R_1 by making transitions to the copy state from the states which can reach some accepting state of M_{R_1} with only one transition. Note that the copy state with a transition to some accepting state comes to have a self-loop, based on this way of construction.

In order to select "repetition of R_1" or "exiting from R_1" deterministically, we

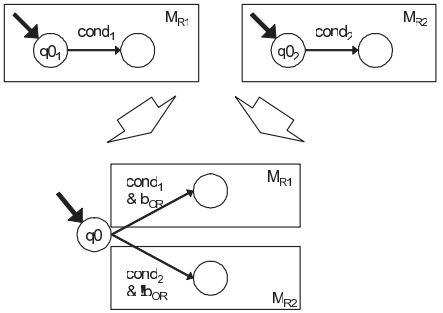

Fig. 8 "R_1 **or** R_2".

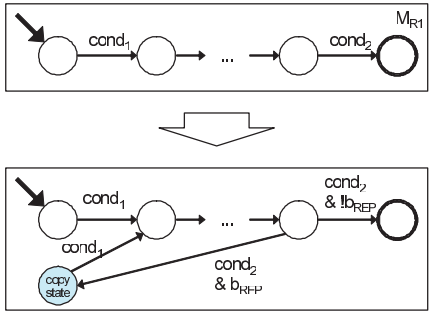

Fig. 9 "R_1 [∗1:$]".

introduce a new logical variable b_{REP}. Let S be the state set from which the automaton can reach one of accepting states with only one transition. For any transition such that $\exists s \in S$, $trans(s, s', cond)$ is true for M_{R_1}, if s' is included in the accepting state set, then the transition condition $cond$ is modified to $cond \wedge \neg b_{REP}$. Otherwise, that is, if s' is the copy state, then the transition condition $cond$ is modified to $cond \wedge b_{REP}$ as shown in **Fig. 9**. Intuitively, if $b_{REP} = 1$ holds, then the automaton selects repetition, and if $b_{REP} = 0$ holds, then the automaton selects exiting the repetition.

For description "R_1 [∗0]" corresponding to the null sequence, a nondeterministic branch also occurs. In order to determinize this nondeterministic branch, we introduce a new logical variable similarly.

© 2009 Information Processing Society of Japan

(d) Concatenation of the two automata

After determinization of the LHS, two automata are concatenated. In syntax S, we have to handle cycle overlapping implication operator `|->`. This is done by merging each pair of the edges of the following two types.

- The edges to the accepting states of the LHS.
- The edges from the initial states of the RHS.

Each of the resulting merged edges comes to have, as its label, the logical conjunction of two logical conditions in the original edges in the LHS and the RHS.

Based on the semantics of the implication operator, all of the failing states in the LHS are changed to the accepting states of the resulting automaton.

(e) Modification of the automaton for handling the "always" property.

At first, we modify the nondeterministic automaton in order to select a starting cycle. Since assertions are required to be checked at *every* cycle, the automaton must start at every cycle. For this purpose, we introduce new logical variables b_{START} and b_{CHECK}. The following shows transition functions for b_{START} and b_{CHECK}, where $next(b)$ indicates the next value of a variable b and $\{a_0, \cdots, a_n\}$ indicates selecting a value a_1, $\cdots$, a_n nondeterministically. The initial value of b_{START} is 0 or 1, b_{CHECK} is 0. We modify the automaton so that only if $b_{START} = 1$ holds, the automaton starts and otherwise, the automaton waits at the initial state.

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

$$next(b_{CHECK}) = \begin{cases} 1 & (b_{START} \vee b_{CHECK} \text{ holds}) \\ 0 & (\text{otherwise}) \end{cases}$$

4.4 Generating Transition Functions

(a) Determinization of the automaton

We determinize the nondeterministic automaton and generate transition functions. For determinization, we adopt one-hot coding.

Let Q be the state set of the automaton. For each state $s \in Q$, we introduce a new logical variable b_s. The following shows a transition function for each logical variable b_s. Here, $S_{prev}(s)$ indicates the state set to which the automaton can reach from the state s with one transition and $cond(s, s')$ indicates the transition condition labeled with the transition from the state s to the state s'. If s is the initial state, then the initial value of a logical variable s is 1, otherwise, the initial value is 0.

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

(b) Addition of substitutions to transition functions

We introduce a storing variable st_v for each local variable v. Note that the records about substitutions, which include a local variable name and substituted expression and a transition (current state, next state and transition condition), are kept at the assertion conversion step. Let $(v, (s, s', cond), e)$ indicate that an expression e is substituted for a local variable v at the transition whose current state is s, next state s' and a transition condition is *cond*. Suppose that we have $lv_i = (v, (s_i, s'_i, cond_i), e_i)$ $(i = 1, \cdots, n)$ for each local variable v. The following shows a transition function for a storing variable st_v corresponding to v.

$$next(st_v) = \begin{cases} e_0 & (b_{s_0} \wedge cond_0 \text{ holds}) \\ e_1 & (b_{s_1} \wedge cond_1 \text{ holds}) \\ & \cdot \\ & \cdot \\ & \cdot \\ e_n & (b_{s_n} \wedge cond_n \text{ holds}) \\ st_v & (\text{otherwise}) \end{cases}$$

Once values of b_{OR}, b_{REP} and b_{START} are fixed at every cycle, only one thread is executed as for the LHS. Thus, at any cycle, only one transition is activated. Therefore, more than one condition cannot be satisfied at the same time in the above transition function.

Finally, we generate a checker as a set of the above transition functions .

4.5 Generating a Temporal Logic Formula

We combine the checker generated in the previous section with a given design, and perform model checking. For model checking, we generate a new property as

© 2009 Information Processing Society of Japan

follows. The following property is described in linear temporal logic (LTL), where "**G** p" indicates that p always must be true. Here, Q, Q_{LHS} and q_F indicate the state set, the state set of the left-hand side of an implication operator and the failing state respectively. Note that the number of failing states is at most one.

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

If this property fails, the given assertion fails for the given design. Otherwise, that is, if this property holds, the given assertion does not fail for the given design.

4.6 Comparison

In the previous work by Long and Seawright [8], logical variables similar to ours are introduced. Their checker, however, does not have the logical variables corresponding to b_{START}, and, as a result, multiple threads are allowed to run. This implies that more than one threads could traverse each single transition edge of a checker at a time. The logical variables are used to choose only one thread at each branch originated from "**or**" operator and "[∗1:$]" operator, once values for the logical variables are fixed at each cycle. **Figure 10** shows how such a logical variable, b_{SEL}, is introduced.

Then, multiple threads can occur, but no threads overlap with each other along any transition edge. The number of threads at a time is limited to the number of transition edges, which is roughly linear to the size of the assertion descriptions. So is the number of necessary storing variables, because each thread may require one storing variable.

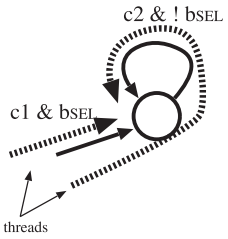

Fig. 10 Thread selection by b_{SEL}.

5. Experimental Results

We implemented our algorithm in C language, and performed experiments for comparing our method with the previous work [8]. As a verification method, we adopt bounded model checking.

5.1 Environment and Examples

The following is our experimental environment:

- CPU: Pentium 4 2.4 GHz
- memory: 2 GB
- OS: Linux 2.4.20-8
- model checker: NuSMV [7]

We checked data consistency for a delay circuit and a FIFO circuit, and measured the verification time and memory requirement. We explain these designs and assertions in the following.

The following shows the behavior of the delay circuit, where the bit width of data is parameterized as *width*.

- Input signal: reqin
- Data: (INPUT) din, (OUTPUT) dout
- Length of delay: 5
- Behavior:
  If reqin = 1 holds, it takes the data "din", and 5 cycles later it outputs this data to "dout".

The following is an assertion for this delay circuit, where x is a local variable. This assertion is for checking input-output data consistency.

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

This can be expressed in syntax S as follows:

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


© 2009 Information Processing Society of Japan

The following shows the behavior of the FIFO circuit, where the bit width of data is parameterized as *width*.

- Input signal: reqin, reqout
- Internal signal: incnt, outcnt (4 bit)
- Data: (INPUT) din, (OUTPUT) dout
- Depth of FIFO: 10
- Behavior:
  If reqin = 1 holds, it takes the data "din" and the tag "incnt", and it increments "incnt".
  One cycle after reqout = 1 holds, it outputs the data to "dout", and it increments "outcnt".

The following is an assertion for this FIFO circuit, where x and *tag* are local variables. This assertion is for checking input-output data consistency.

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

This can be expressed in syntax S as follows:

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

We checked the delay circuit and the FIFO circuit using bounded model checking, where the bound of the bounded model checking is parameterized as k.

For the delay circuit and FIFO circuit, each parameter is set as follows. For each setting, we measured the verification time and memory requirement.

- bound k: 10, 15
- bit width *width*: 4, 5, 6, 7, 8 bit

5.2 Experimental Results

Table 1 and **Table 2** show the verification time for the delay circuit and the FIFO circuit respectively.

Table 3 and **Table 4** show the memory requirement for the delay circuit and the FIFO circuit respectively. The memory requirement was improved by 10-30% for both the delay circuit and the FIFO circuit. The number of storing variables is reduced in our method compared with the previous work [8], as shown

Table 1 Verification time (sec) for the delay circuit.

k	*width*	ours	Ref. 8)	improvement
10	4	0.6	0.7	14%
	5	1.4	1.7	18%
	6	3.5	4.2	17%
	7	11.0	13.1	16%
	8	35.2	38.3	9%
15	4	2.0	2.2	10%
	5	5.3	6.0	12%
	6	16.6	16.7	1%
	7	55.9	56.0	0%
	8	188	188	0%

Table 2 Verification time (sec) for the FIFO circuit.

k	*width*	ours	Ref. 8)	improvement
10	4	23.7	29.1	19%
	5	62.2	73.5	15%
	6	184	211	13%
	7	345	421	18%
	8	1022	1215	16%
15	4	144	258	46%
	5	300	490	39%
	6	804	1101	27%
	7	2350	3214	27%
	8	8167	10308	21%

Table 3 Memory requirement (MB) for the delay circuit.

k	*width*	ours	Ref. 8)	improvement
10	4	11.6	13.1	12%
	5	14.4	17.4	18%
	6	19.7	25.9	24%
	7	31.0	42.9	28%
	8	52.7	77.0	32%
15	4	13.1	15.2	14%
	5	19.3	21.6	11%
	6	25.6	34.1	25%
	7	42.7	59.1	28%
	8	77.9	114	32%

© 2009 Information Processing Society of Japan

Table 4 Memory requirement (MB) for the FIFO circuit.

k	*width*	ours	Ref. 8)	improvement
10	4	26.3	31.1	16%
	5	41.3	48.3	15%
	6	70.2	83.0	16%
	7	129	154	17%
	8	252	293	14%
15	4	40.2	62.1	36%
	5	66.1	88.6	26%
	6	117	160	27%
	7	219	275	21%
	8	450	600	25%

Table 5 Total number of logical variables.

k	*width*	ours	Ref. 8)	improvement
10	4	4	20	80%
	5	5	25	80%
	6	6	30	80%
	7	7	35	80%
	8	8	40	80%
15	4	8	32	75%
	5	9	36	75%
	6	10	40	75%
	7	11	44	75%
	8	12	48	75%

in **Table 5**.

The number of the logical variables is equal to the bit width of the local variable. In Table 5, the sum of the number of logical variables for all storing variables is shown as total number of logical variables. In our method, since one storing variable is allocated for each local variable, the total number of logical variables is equal to the sum of the bit width of each local variable. In the previous work [8], since the number of storing variable for each local variable is equal to the sum of the number of "**##**1" operator and "[∗1:$]" operator and, as a result, the total number of logical variables is equal to n times of the sum of the bit width of each local variable, where n is equal to the sum of the number of "**##**1" operator and "[∗1:$]" operator. In this experiment, $n = 5$ for the delay circuit, $n = 4$ for the FIFO circuit. The bit width of the local variable *tag* in the FIFO circuit is 4 bit. The total number of logical variables is improved by 75–80% compared with the previous work [8].

5.3 Consideration

As compared with the previous approach [8], the reduction of the number of variables is relatively large, but the improvement of performance is rather limited.

The Long and Seawright's checker allows multiple threads to run at a time for each of possible combinations of values on newly introduced variables. The bounded model checker, using some SAT solver, virtually checks all of the multiple threads one by one. On the other hand, our checker allows only one thread to run at a time by introducing b_{START}, which chooses a starting cycle nondeterministically. The bounded model checker checks one thread at a time for each of the all possible starting cycles.

This suggests that the number of computational steps necessary for bounded model checking does not change much in both of the approaches. We think that this is the reason that we can achieve only small or negligible improvement in terms of run-times.

In particular, the improvement ratio worsens, when the bit width increases. We think this is because the major bottleneck of computational complexity shifts from the assertion to the design, when the bit width increases.

On the other hand, our algorithms always generates shorter descriptions of transition functions than the Long and Seawright's algorithm. As a result, shorter boolean formulas are generated inside the bounded model checker. In other words, many distinct sub-formulas which occur in the Long and Seawright's approach are represented by a fewer number of sub-formulas in our approach. We think this is the reason that we can achieve better and stable improvement in memory requirement.

6. Conclusion

We showed a checker generation algorithm for SVA assertions with local variables, in which the number of storing variables for each local variable is only one. We also show the experimental results for a delay circuit and a FIFO circuit. Compared with the previous work [8], the memory requirement was improved by 10–30%. As we discussed in Section 5.3, the number of computational steps does not change much in our approach. The main contribution of our paper is

© 2009 Information Processing Society of Japan

improvement in terms of memory requirement.

Future works include extending the syntax assertions. In particular, handling "intersect" operator efficiently would be important from a practical point of view. Furthermore, in order to improve run-time performance, we could try some coding method other than one-hot coding.

We also think that our approach can be applied to EUF-based model checking [14], in which data and an operation are abstracted, respectively, by a variable and a function symbol in the first-order logic.

References

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) Clarke, Jr., E.M., Grumberg, O. and Peled, D.A.: Model Checking, The MIT Press (1999).
4) McMillan, K.L.: Symbolic Model Checking, Kluwer Academic Publishers (1993). Some Complexity Results for System Verilog Assertions, Computer-Aided Verification, LNCS 4144, pp.205–218 (2007).
5) Foster, H., Krolnik, A. and Lacey, D.: Assertion-based Design, 2nd Ed., Springer (2004).
6) Biere, A., Cimatti, A., Clarke, E.M., Fujita, M. and Zhu, Y.: Symbolic Model Checking Using SAT Procedures instead of BDDs, Proceedings of Conference on Design Automation, Vol.98-2, pp.142–170 (1992).
7) Cimatti, A., Clarke, E., Giunchi, F. and Roveri, M.: NuSMV: A New Symbolic Model Verifier, Proceedings of International Conference on Computer-Aided Verification (1999).
8) Long, J. and Seawright, A.: Synthesizing SVA Local Variables for Formal Verification, Proceedings of Design Automation Conference, pp.75–80 (2007).
9) Das, S., Monhanty, R., Dasgupta, P. and Chakrabarti, P.P.: Synthesis of SystemVerilog Assertions, Proceedings of Design Automation and Test in Europe, pp.70–75 (2006).
10) Morin-Allory, K. and Borrione, D.: Proven Correct Monitors from PSL Specifications, Proceedings of Design Automation and Test in Europe, pp.1246–1251 (2006).
11) Bustan, D. and Havlicek, J.: Some Complexity Results for System Verilog Assertions, Computer-Aided Verification, LNCS 4144, pp.205–218 (2007).
12) Takeuchi, S., Hamaguchi, K. and Kashiwabara, T.: Encoding Assertions with Dynamic Local Variables for Bounded Property Checking, Synthesis and Simulation Meeting and International Interchange, pp.515–521 (2007).
13) Ullman, J.D., Hopcroft, J.E. and Motwani, R.: Introduction to Automata Theory, Languages and Computation, 3rd Ed., Addison-Wesley (2006).
14) Bryant, R.E., German, S. and Velev, M.N.: Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions, Computer-Aided Verification, LNCS 2404, pp.78–92 (2002).

(Received May 23, 2008)
(Revised August 22, 2008)
(Accepted October 9, 2008)
(Released February 17, 2009)

(Recommended by Associate Editor: *Yusuke Matsunaga*)

Sho Takeuchi received a master of degree in information science and technology, Osaka University, Japan in 2008. He is currently with Hitachi Government and Public Corporation System Engineering Corp.

Kiyoharu Hamaguchi received the B.E., M.E. and Ph.D. degrees in information science from Kyoto University, Japan, in 1987, 1989 and 1993 respectively. In 1994, he joined the Department of Information Science, Kyoto University. He is currently with Graduate School of Information Science and Technology, Osaka University as Associate Professor. His current interests include formal verification and computer aided design.

© 2009 Information Processing Society of Japan

Toshinobu Kashiwabara received the B.E., M.E. and Dr.Eng. degrees from Osaka University, Japan, in 1969, 1971 and 1974 respectively. He joined the faculty of Osaka University in 1974, and is currently a Professor at Graduate School of Information Science and Technology. His research interests include circuit layout and design of combinatorial algorithms. He is a member of IEEE.

© 2009 Information Processing Society of Japan