Synthesis of System Verilog Assertions

Sayantan Das Rizi Mohanty Pallab Dasgupta P.P. Chakrabarti
Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur, India.

Abstract

In recent years, Assertion-Based Verification is being widely accepted as a key technology in the pre-silicon validation of system-on-chip(SOC) designs. The System Verilog language integrates the specification of assertions with the hardware description. In this paper we show that there are several compelling reasons for synthesizing assertions in hardware, and present an approach for synthesizing System Verilog Assertions (SVA) in hardware. Our method investigates the structure of SVA properties and decomposes them into simple communicating parallel hardware units that together act as a monitor for the property. We present a tool that performs this synthesis, and also show that the chip area required by the monitors for a industry standard ABV IP for the ARM AMBA AHB protocol is quite modest.

1. Introduction

Assertion-Based Verification (ABV) is assuming a significant role in the design validation flow of chip design companies. In recent times active participation from the design and EDA industries have led to the adoption of several formal languages for assertion specification. These include Forspec (of Intel) [2], Sugar/PSL (of IBM/Accelera) [7] and OVA (of Synopsys) [5]. Several companies are developing / marketing ABV verification IPs for standard protocols, like PCI Bus [6], ARM AMBA Bus [1], and Hypertransport [3].

More recently, System Verilog [9] has integrated the specification of assertions with the core hardware description language. The intent of specifying assertions within the design is to enable the simulator to check the assertions during simulation. Thus, the assertions are not treated as part of the synthesizable Verilog code, rather they are treated as properties that are expected to hold on the design.

This paper studies the problem of synthesizing SVA checkers in hardware. There are three interesting reasons that motivate us to investigate this, namely:

1. To extend ABV to hardware emulation and early design prototypes (such as FPGA),
2. For debugging post-silicon violations of temporal properties, and
3. To enable the System Verilog designer to design circuit components that are triggered by complex temporal behavior of the other components.

The first of the above requirements comes mainly from ASIC designers who regularly create early prototypes of their design and use emulation to beat the simulation bottleneck in validation. It is usually very difficult to debug violation of complex temporal properties during emulation, which is one of the motivating reasons to study the option of synthesizing SVA checkers in hardware.

The second requirement will increasingly become important in large (and costly) designs, because companies want to debug the reasons for failure in specific situations. Currently finding out the violation of a temporal event after tapeout is an extremely complex problem. Built-in checkers will provide valuable insights to the cause of failure.

We believe that the third requirement will be a key issue in future design practices. In many safety-critical applications, the designer would like the design to recover from (static or intermittent) temporal faults if they exist. This can be achieved by enabling the triggering of the recovery module on a match of the checker, modeling the temporal fault.

This paper investigates the problem of synthesizing monitors for SVA. Our objective is *not* to synthesize a universal checker that accepts any SVA property and monitors it, but to create dedicated monitors for the given properties. The advantage of creating monitors in hardware is that the monitors for individual properties can work in parallel and hence does not suffer from the state-explosion problem.

It may be pointed out that there has been previous attempts to use pre-defined monitors. Notable among these are the OVL library [11, 4] and IBM's FOCs [8]. However none of these papers presents the details of synthesizing SVA.

2. SVA Introduction

This section outlines the structure of SVA. The building blocks of SVA are called *Sequence Expressions(SE),*

3-9810801-0-6/DATE06 © 2006 EDAA

that are used to describe the temporal behavior of a system. The most basic sequence expressions are the signals and Boolean expressions over the signals. Temporal sequence expressions can be constructed by using the *time range* operators. The syntax of a SE is defined as follows.

- SE $\longrightarrow$ SE TIME_RANGE SE | SE BOOLEAN_ABBRV | SE SEQUENCE_OP SE | first_match(SE) | EXP throughout SE | EXP | (SE).
- TIME_RANGE $\longrightarrow$ ##k | ##[$k_1 : k_2$]
- BOOLEAN_ABBRV $\longrightarrow$ [*k] | [*$k_1 : k_2$] | [*>$k_1 : k_2$] | [*=$k_1 : k_2$]
- SEQUENCE_OP $\longrightarrow$ and | or | intersect | within.
- EXP $\longrightarrow$ EXP || EXP | EXP && EXP | !EXP | p, p is a Boolean signal

The structure of a SVA property is as follows:

PROPERTY $\longrightarrow$ **property** PROP_EXP **endproperty**

where PROP_EXP can be of two types, namely:

[CLOCK_EVENT] [**disableiff** EXP] [**not**] SE
$|\rightarrow$ [**disableiff** EXP] [**not**] SE or,
[CLOCK_EVENT] [**disableiff** EXP] [**not**] SE
$|\Rightarrow$ [**disableiff** EXP] [**not**] SE

In the above forms:

- **CLOCK_EVENT** represents the name of the clock against which the property is evaluated.
- **disableiff EXP** allows the user to specify asynchronous reset. If the EXP becomes true then the evaluation stops and the property is accepted as true.
- The **not** operator before a sequence expression *s* implies that whenever *s* matches, **not** *s* fails and vice-versa.
- $|\Rightarrow$ and $|\rightarrow$ are implication operators differing by the fact that the start match of the consequent part may start at the same time stamp at which the antecedent matches or one time stamp later, depending on the implication operator being $|\rightarrow$ or $|\Rightarrow$ respectively.

The Appendix 3 in [9] suggests that the operators *or,intersect, [*$N_1 : N_2$],##[$N_1 : N_2$]* and *first_match* are sufficient to describe any sequence expression in SVA.

2.1. Distribution of disjunction

Sequence expressions that have the semantics of disjunction can give out of order match i.e. the match corresponding to a given start may appear after the match of a later start. Consider the following sequence expression $\mathcal{S}$ = ((s_1 *or* s_2) *intersect* s_3). Confirming to the semantics of *intersect* $\mathcal{S}$ will match if one of s_1 or s_2 and s_3 matches at the same time. The occurrence of a match of s_3 and one of s_1 or s_2 is not a sufficient condition to determine a match of $\mathcal{S}$ because the two match outputs might correspond to two different start signals. Storing the corresponding start signal for each match is impractical because this would require the information to be stored in all the states of the checker leading to large memory requirement for synthesizing them. Thus in absence of this information it is impossible to determine whether the match outputs of (s_1 *or* s_2) and s_3 corresponds to the same start signal unless the match outputs are ordered. Thus we define the flattening function $\mathcal{F}$(s) to remove the disjunctions which causes the above problem.

1. $\mathcal{F}$(B) $\equiv$ B //Where B is boolean
2. $\mathcal{F}(s_1$ ##k $s_2) \equiv \mathcal{F}(s_1)$ ##k $\mathcal{F}(s_2)$
3. $\mathcal{F}(s_1$ ##[$k_1 : k_2$] $s_2) \equiv \mathcal{F}(s_1$##$k_1 s_2)$ or. . .or $\mathcal{F}(s_1$##k_2 $s_2)$
4. $\mathcal{F}(s_1$ ##[k_1 : \$] $s_2) \equiv \mathcal{F}(\mathcal{F}(s_1)$ ##[k_1 : \$] $\mathcal{F}(s_2))$
5. $\mathcal{F}(s_1$[*k]) $\equiv \underbrace{\mathcal{F}(s_1)\#\#1\mathcal{F}(s_1)\#\#1 \ldots \#\#1\mathcal{F}(s_1)}_{k\ times}$
6. $\mathcal{F}(s_1[*k_1 : k_2] \equiv \mathcal{F}(s_1$ [*k_1]) or . . . or $\mathcal{F}(s_1$ [*k_2])
7. $\mathcal{F}(s_1[*k_1 :$ \$] $\equiv \mathcal{F}(\mathcal{F}(s_1)$ [*k_1 : \$])
8. $\mathcal{F}((s_1$ or $s_2)$##[k_1:k_2]$s_3) \equiv$ ($\mathcal{F}(s_1)$ ##[k_1:k_2] $\mathcal{F}(s_3)$) or ($\mathcal{F}(s_2)$ ##[k_1:k_2] $\mathcal{F}(s_3)$)
9. $\mathcal{F}((s_1$ or $s_2)[*k_1$:$k_2]) \equiv \mathcal{F}(s_1)$ [$*k_1$:k_2] or $\mathcal{F}(s_2)$ [$*k_1$:k_2]
10. $\mathcal{F}((s_1$ or $s_2)$ intersect $s_3) \equiv$ ($\mathcal{F}(s_1)$ intersect $\mathcal{F}(s_3)$) or ($\mathcal{F}(s_2)$ intersect $\mathcal{F}(s_3)$)
11. $\mathcal{F}$*(first_match(s))* $\equiv$ *first_match*($\mathcal{F}(s)$)

In rules 8-10, if s_3 contains an *or* operator, then the distribution is symmetrical. In case of *intersection* operator if one of it's operand is of bounded and the other is unbounded then the unbounded length sequence expression can be decomposed appropriately to match the bounded length operand of the intersection. For example (a ##[2 : \$] b) *intersect* (c ##4 d) can be equivalently written as ((a ##4 b) intersect (c ##4 d)) as other possibilities will always resolve to false. Also for similar reason (a *[2 : \$] ##1 b) *intersect* (c ##4 d) can be equivalently re-written as (a *[3] ##1 b) *intersect* (c ##4 d) as other possibilities will always resolve to false. Note that recursive application of $\mathcal{F}$ ensures that the sequence expression operands of $intersect$ are disjunction free and hence will always give in-order match.

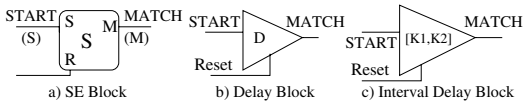

Figure 1. Basic sequence expression block

3. Sequence Expression Synthesis Algorithm

We use a *divide-and-conquer* approach for synthesis of sequence expression. The basic idea is to break the sequence expressions as a sequence of expressions concatenated with the corresponding time range expressions. The checkers recognizing these smaller sequence expressions

are then interconnected so that they communicate among each other to determine a match or fail of the actual sequence expression. Every checker generated by our algorithm (as shown in Fig 1 (a)) has an input, *start*(S), which triggers the start of checking and a single output *match*(M) which indicates the match of an expression. There is also a reset input for each block, on a high reset each block moves to it's initial state and waits for the start. We also have a DelayFSM(D) block as shown in Fig 1 (b), having a single input, start(S), a single output, match(M), and a delay parameter, D. On receiving the start input this block waits for D cycles and then asserts the match output. Another variant of this block, called the IDelayFSM block is shown in Fig 1 (c), is parameterized by a delay interval, $[k_1, k_2]$. On receiving the start input this block waits for k_1 cycles and then asserts the match output for the next $k_2 - k_1$ cycles. If k_2 is \$ then the block will hold the match output high till the end of simulation after the first k_1 cycles. Both these blocks are synthesized as finite state machines. We define a function $\mathcal{L}$(s), which returns the lower bound on the number of time steps required by a sequence expression s to match.

We have divided the total set of SVA into 4 sub groups namely, *Simple Sequence Expression* (SSE), *Interval Sequence Expression* (ISE), *Complex Sequence Expression* (CSE) and *Unbounded Sequence Expression* (USE).

3.1. Synthesizing SSE and ISE

This subsection defines SSE and ISE and describes the algorithms for synthesizing them. SSE and ISE are sequence expressions formed by Boolean expressions, TIME_RANGE operator and the SEQUENCE_OP only.

The following two algorithms outlines our method for synthesizing SSE and ISE. The reader may refer to [9] for the detailed semantics of these operators. In all the Figures referred in our synthesis algorithms 'S' represent the start input and 'M' represents the match output.

Algorithm 1 (h) SynthSSE(s: *SSE*)

If s = s_1 $\langle op \rangle$ s_2 where s_1 and s_2 are SSE, we use M_1 and M_2 to denote SynthSSE(s_1) and SynthSSE(s_2) respectively.
case s = EXP //EXP: a Boolean expression
s is synthesized as a combinational block.
case s = s_1 **##k** s_2
let M_3 = DelayFSM(k). M_1 identifies a match of s_1 and triggers M_3, which then gives out a match k cycles later to trigger M_2. M_2 identifies the match of s_2. The match output of M_2 is used as the match output of the checker.
case s_1**[*k]**
synthesize k copies of M_1 namely $M_{11}, M_{12}, \ldots, M_{1k}$. The M_{1i} block is then connected to the M_{1i+1} block using a DelayFSM(1) block. The match output of M_{1k} is used as the match output of the checker.

Algorithm 2 (h) SynthISE(s: *ISE*)

If s = s_1 $\langle op \rangle$ s_2 where s_1 and s_2 are ISE, we use M_1 and M_2 to denote SynthISE(s_1) and SynthISE(s_2) respectively.
case s = s_1 **##[**k_1 **:** k_2**]** s_2
M_3=IDelayFSM(k_1, k_2). M_1 identifies a match of s_1 and triggers M_3,which waits for k_1 cycles and then gives out matches for the next $k_2 - k_1$ cycles each triggering M_2. The match of M_2 is used as the match output of the checker.
case s = s_1**[***k_1 : k_2**]**
create k_2 copies of the block M_1 and connect them in the same manner like the checker creation for $s_1[*k_2]$. However,the final match output of the checker is obtained by connecting the match outputs of the last $(k_2 - k_1)$ blocks by an OR-gate.

3.2. Synthesizing CSE

CSE consists of sequence expressions containing operators *or, first_match,* and *intersect.* However CSE is a proper super set of SSE and ISE. In this algorithm we assume that the sequence expressions that appear to the left or right of the *intersect* are all SSE. *Sequence expressions* which defines temporal behavior over unbounded time are synthesized differently and is explained later in the Subsection 3.3.

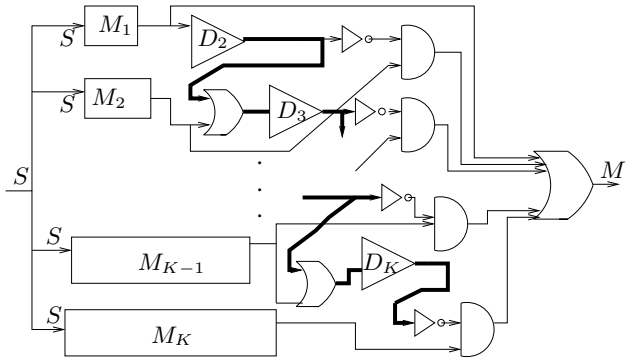

Figure 2. $first\_match(s)$

Algorithm 3 (h) SynthCSE(s: *CSE*)

If s = s_1 $\langle op \rangle$ s_2 where s_1 and s_2 are CSE, we use M_1 and M_2 to denote SynthCSE(s_1) and SynthCSE(s_2) respectively.
case s = s_1 **or** s_2
The sequence expression s matches whenever one of s_1 or s_2 matches. This is done by connecting the match outputs of M_1 and M_2 with an OR-gate.
case s = s_1 **intersect** s_2
The sequence expression s matches when s_1 and s_2 matches at the same time. This is achieved by connecting the match outputs of M_1 and M_2 with an AND-gate.

case s = first_match(s')
first_match(s') $\equiv$ first_match($\mathcal{F}(s')$) $\equiv$ first_match(s_1 or s_2 or . . . or s_k) where each sequence expression s_i is free from disjunctions and $\mathcal{L}(s_1) \leq \mathcal{L}(s_2) \ldots \leq \mathcal{L}(s_k)$. Confirming to the semantics of first_match a match of s will correspond only to the match of s_i having the least $\mathcal{L}(s_i)$. This functionality is achieved by making each s_i issue a match iff it matches and all s_j $\forall j < i$ fails. This is done by the checker M_i as shown in Fig 2. In Fig 2 the checker M_i and M_{i+1} are connected using a Delay Block D_{i+1} with a delay parameter equal to $(L(s_{i+1}) - L(s_i))$. Whenever M_i matches and the output of its corresponding Delay Block D_{i+1} is zero(i.e none of the previous blocks have matched) the final match output is asserted. This is done for every checker M_i. Also whenever M_i identifies a match, this information is propagated through the delay block D_{i+1} to indicate the machine $M_j(j > i)$ that a match has already been identified. The bold line in the Fig 2 indicates this path.

3.3. Synthesizing USE

This sub-section addresses the problem of synthesizing sequence expressions which expresses unbounded temporal behavior. But first we identify expressions which cannot be synthesized in bounded area.
case s = $s_1\#\#[k_1 : \$]\ s_2$ **intersect** $s_3[*k_2 : \$]$
The expression s will match when the right hand and the left hand side of the *intersect* operator matches at the same time and also they correspond to the same start signal. The sequence expression $s_1\#\#[k_1 : \$]\ s_2$ can take arbitrary large time to match after the arrival of a start signal and in the meanwhile there might be arbitrarily large number of matches of $s_3[*k_2 : \$]$ corresponding to different start inputs and hence it is practically impossible to store which of these matches correspond to which start. Thus, when s_2 and $s_3[*k_2 : \$]$ matches at the same time, it is impossible to determine if they correspond to the same start input implying s cannot be synthesized in bounded area.
first_match($s_1\#\#[k_1 : \$]\ s_2$ **intersect** $s_3[*k_2 : \$]$**)**
This expression is not synthesizable because ($s_1\#\#[k_1 : \$]$ s_2 intersect $s_3[*k_2 : \$]$) is not synthesizable.

Algorithm 4 (h) SynthUSE(s: *USE*)

If s = $s_1\langle op\rangle s_2$ *where* s_1 *and* s_2 *are USE, we use* M_1 *and* M_2 *to denote SynthUSE(*s_1*) and SynthUSE(*s_2*) respectively.*
case s = s_1 **##[**k_1 **: $]** s_2
Let M_3 = IDelayFSM(k_1,$). M_1 identifies a match of s_1 and triggers M_3,which gives out a match k_1 cycles later and thereafter holds it high forever. The match of M_3 triggers the checking of M_2. M_2 identifies the match of s_2. The match output of M_2 is also the match output of the checker.
case s = s_1**[*k_1 **: $]**
Create k_1 copies of M_1 and connect them in a manner identical to the construction of $s_1[*k_1]$. The match output of the checker is connected with the match output of the k^{th} M_1 block. The output of k^{th} M_1 block is OR-ed with the output of the $(k-1)^{th}$ M_1 block and connected with the start input of the k^{th} M_1 block. This connection ensures match of s due to match of $s_1*[k]$ where $k \geq k_1$.
case s=(s_1 **##[**k_1 **: $]** s_2**) intersect (**s_3 **##[**k_2 **: $]**$s_4$**)**
Here we assume $(s_1\ and\ s_3)$ is synthesizable. The sequence expression will match when both s_2 and s_4 matches at the same time. However the two matches must correspond to the same start input. In order to check this, we introduce an extra redundancy in the checker in form of a checker for $(s_1\ and\ s_3)$. Note that for s to match $s_1\ and\ s_3$ must match at some previous time. Let l be the minimum time required for s to match after $s_1\ and\ s_3$ has matched. Here both s_1 and

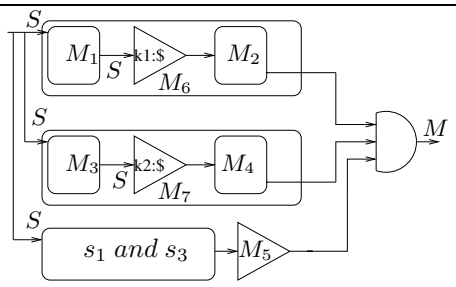

Figure 3. (s_1**##[**k_1 **: $]**$s_2$**)intersect(**$s_3$**##[**$k_2$ **: $]**$s_4$**)**

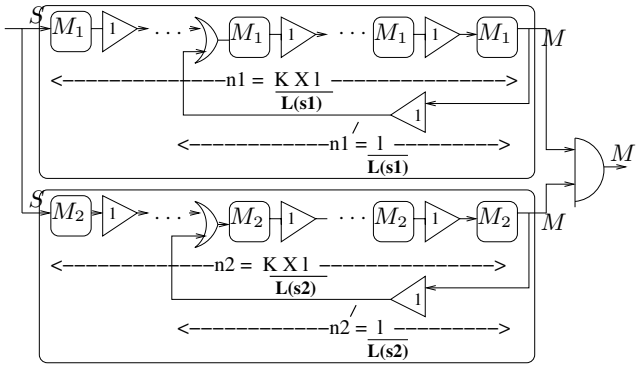

Figure 4. (s_1 **[*k_1 **: $]**)**intersect(**$s_2$ **[*k_2 **: $]**)**

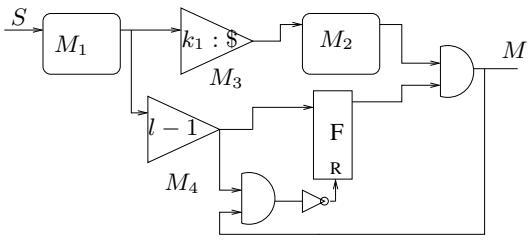

Figure 5. first_match(s_1 **##[**k_1 **: $]** s_2**)**

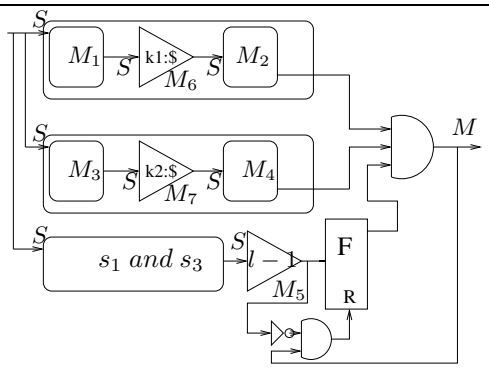

Figure 6. first_match((s_1 **##[**$k_1 : \$]s_2$**) intersect (**$s_3$ **##[**$k_2 : \$]s_4$**))**

s_3 are SSE and let l = $\mathcal{L}(s)$ - MAX($\mathcal{L}(s_1), \mathcal{L}(s_3)$). Consider a match of ($s_1$ *and* s_3) at time t corresponding to a start input S. Now any match of s_2 and s_4 after or on time $t + l$ will correspond to a match of s corresponding to the start S. Keeping this intent in mind our synthesis algorithm is as follows: Let M_5=IDelayFSM(l,$),$M_6$=IDelayFSM($k_1$, $), M_7=IDelayFSM(k_2, $). The machine M_5 is triggered by a match of s_1 *and* s_3, a match of M_5 indicates that s_1 *and* s_3 have matched l cycles earlier. The outputs of M_2,M_4 and M_5 are connected by an ANDgate whose output comprise the final match output as shown in Fig 3.

case s = (s_1 **[***$k_1 : \$]$**) intersect (**$s_2$ **[***$k_2 : \$]$**)**

Here we assume that s_1 and s_2 are synthesizable. Let l be the Lowest Common Multiple(LCM) of $\mathcal{L}(s_1)$ and $\mathcal{L}(s_2)$ and k the smallest integer such that $(k \times l) \geq (k_1 \times \mathcal{L}(s_1))$ and $(k \times l) \geq (k_2 \times \mathcal{L}(s_2))$. Sequence expression s (if matches) will match for the first time at time step $k \times l$. Subsequent matches for s(if occurs) will arrive at time steps equal to multiple of l after the first match. Keeping this intent in mind the checker is synthesized as follows: Let $n_1 = \frac{k \times l}{\mathcal{L}(s_1)}$, $n_2 = \frac{k \times l}{\mathcal{L}(s_2)}$, $n_1' = \frac{l}{\mathcal{L}(s_1)}$ and $n_2' = \frac{l}{\mathcal{L}(s_2)}$. Synthesize n_1 blocks of M_1 and n_2 blocks of M_2 and connect them as shown in Fig 4. The feedback loop from the ${n_1}^{th}$ M_1 block to the $(n_1 - n_1' + 1)^{th}$ M_1 block and the ${n_2}^{th}$ M_2 block to the $(n_2 - n_2' + 1)^{th}$ M_2 block is necessary to identify subsequent matches of s. The match output of the checker is connected with the match output of the ${n_1}^{th}$ M_1 and the ${n_2}^{th}$ M_2 block by an AND gate.

case s = first_match(s')

if(s' is of the form ($s_1[*k_1 : \$]$) or of the form ($s_1[*k_1 : \$]$ **or** $s_1[*k_2 : \$]$)
Replace $s_i[*k_i : \$]$ by s[$*k_i$] and synthesize
endif
if s' ≡ (**s**$_1$**[***$k_1 : \$$**] intersect** **s**$_2$**[***$k_2 : \$$**])**
Replace $s_i[*k_i : \$]$ by $s_i[*\frac{k \times l}{\mathcal{L}(s_i)}]$ and synthesize
l = LCM($\mathcal{L}(s_1), \mathcal{L}(s_2)$) and k is the smallest integer such that k × l > $k_i \times \mathcal{L}(s_i)$ $\forall i = 1, 2$
endif

case s = first_match(s_1 $\#\#[k_1 : \$]$ s_2**)**

Here we assume s_2 to be a SSE otherwise we use $\mathcal{F}$ to decompose it into SSEs. Let M_3 = IDelayFSM(k_1,$) and l be the minimum time required for s to match after a match of s_1 (l = $\mathcal{L}(\#\#[k_1 : \$]\ s_2)$). Let M_4 = DelayFSM($l-1$). The output of M_4 is connected to the input of a flipflop F. The *sequence expression* s will match when M_2 matches and F is high. The reset input of F is asserted when s matches but M_4 does not. The reset input ensures that there are no multiple matches for a single start. Fig 5 shows the above connection. It should be noted that if s_1 contains unbounded temporal operators then we will synthesize M_1 to be a checker for recognizing first_match(s_1) instead of s_1.

case s = first_match((s_1 **##[**$k_1 : \$]s_2$**) intersect (**$s_3$ **##[**$k_2 : \$]s_4$**))**

The synthesis of this checkers is almost the same as the synthesis of (s_1 [*$k_1 : \$$]) $intersect$ (s_2 [*$k_2 : \$$]). The only differences are that we change M_5 to DelayFSM(l-1) and store the output of M_5 in a flipflop F. The reset input of F is asserted when there is a match of s but no match of M_5. The reset input ensures that there are no multiple matches for a single start. Fig 6 shows the above connection.

Until now we have only provided algorithms to synthesize checkers for *sequence expressions* which asserts match when the *sequence expression* matches. But how to identify the fail? We have solved this problem as follows. Given a *sequence expressions* s we create a checkers corresponding to *not s* such that the match of *not s* correspond to a fail of s. Below we provide the rules to generate *not s* from s.

1. not(exp) ≡ ¬exp // Where exp is Boolean expression.
2. not(s_1 ##k s_2) ≡ not(s_1) or s_1 ## k not(s_2)
3. not(s_1 ##[$k_1 : k_2$] s_2) ≡ not(s_1) or (s_1 ## k_1(not(s_2) and ##1 not(s_2) and . . . and ##($k_2 - k_1$) not(s_2)))
4. not(s_1 ##[$k_1 : \$$] s_2) ≡ not(s_1)
5. not (s_1[*k_1]) ≡ not(s_1) or s_1 ##1 not(s_1[*($k_1 - 1$)])
6. not (s_1[*$k_1 : k_2$]) ≡ not(s_1[*k_1])
7. not first_match(s) ≡ not(s)
8. not(s_1 or s_2) ≡ not(s_1) and not(s_2)
9. not(s_1 intersect s_2) ≡ not(s_1) or not(s_2), s_1 and s_2 are SSE.
10. not ((s_1 [*$k_1 : \$$]) intersect (s_2 [*$k_2 : \$$])) ≡ not(s_1) or not(s_2) or not(s_1 [*l_1] intersect s_2 [*l_2]). Where, l_1 = $N \times \frac{LCM(\mathcal{L}(s_1), \mathcal{L}(s_2))}{\mathcal{L}(s_1)}$, l_2 = $N \times \frac{LCM(\mathcal{L}(s_1), \mathcal{L}(s_2))}{\mathcal{L}(s_2)}$
N is the smallest integer such that $N \times LCM(\mathcal{L}(s_1), \mathcal{L}(s_2))$ is greater than both $k_1 \times \mathcal{L}(s_1)$ and $k_2 \times \mathcal{L}(s_2)$

4. SVA Property Synthesis

We synthesize all the sequence expression blocks of a given SVA property, and then use these blocks to synthesize the property. The following algorithm outlines our methodology for synthesizing SVA properties using the blocks for the sequence expressions (given by Algorithms SynthSSE,SynthISE,SynthCSE and SynthUSE).

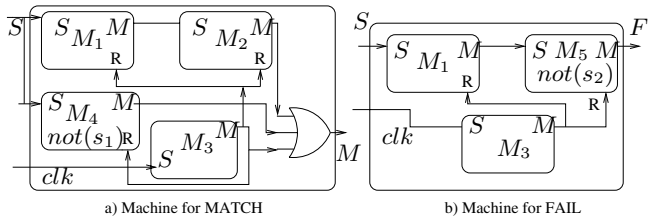

Figure 7. SVA synthesis

Algorithm 5 (h) SynthSVA(p: *SVA property*)

case p = (clk_exp) disableiff exp [not] $s_1 \mid\!\rightarrow$**[not]** s_2
Let M_1 = SynthSVA(s_1), M_2 = SynthSVA(s_2), M_3 = SynthSVA(not(exp)) and M_4 = SynthSVA($not(s_1)$. If M_3 matches then the property evaluation stops and a match is given to the output. Stopping a evaluation is achieved by connecting the match output of M_3 with the reset input of M_1,M_2 and M_4. A match of M_4 makes the property match vacuously. The property is synthesized by interconnecting the blocks M_1,M_2,M_3 and M_4 as shown in Fig 7(a). The property will fail(F) when s_1 matches but s_2 fails. In order to identify the fail of the property we synthesize a new block M_5=SynthSVA(not(s_2)) and then connect M_1,M_3 and M_5 as shown in Fig 7(b).
case p = (clk_exp) disableiff exp [not] $s_1 \mid\!\Rightarrow$**[not]** s_2 Synthesis of this property is identical to that of the previous one except that the blocks corresponding to s_1 and s_2 are connected through a DelayFSM(1) instead of a wire.
If the **not** operator is present before a sequence expression s, then we synthesize *not(s)* instead of s.

5. Results

Our tool decomposes a given set of properties into the basic sequence expression blocks and translates them into synthesizable Verilog. The tool implements the interconnections between these blocks by instantiating these basic blocks within higher-level modules.

We used an industry standard Assertion based verification IP for the ARM AMBA AHB [1] as a case study. We synthesized the assertions in this suite and simulated this code along with the models for the AMBA AHB models using the Synopsys VCS simulator. We compared the output of this simulation with the output of VCS simulation of the same models with the assertion monitoring done by the Synopsys OVA checker. The outputs were identical, indicating that the synthesis produced the correct monitors. Curiously, we found that VCS was able to simulate the synthesized checkers faster than the OVA checker. Table 1 compares the run times for the AMBA AHB master, slave and arbiter properties. The table also shows the number of inputs and outputs of the AHB interfaces, and the number of assertions in the VIP.

We then used the Synopsys Design Compiler [10] to estimate the area required by the checkers using the 0.18μ Synopsys library. The table shows the number of ports, nets, and cells used by our circuits, along with the total combinational and sequential area. The last column in this table shows that the area overhead for the on-chip checkers is quite modest for a VIP which is quite complex by industry standards.

Circuit Module	No of inputs	No of outputs	No of assertions	Time OVA(s)	Time OUR(s)
master	11	9	14	1.4	1.12
slave	13	4	6	1.2	1.09
arbiter	14	3	8	1.29	1.19

Circuit Module	#ports	#nets	#cells	Area	% increase in Area
master	5	44	41	9000	15%
slave	6	32	21	4300	8%
arbiter	5	36	31	2150	12%

Table 1. Experimental Results on AHB

Acknowledgments

Pallab Dasgupta and P.P.Chakrabarti acknowledge the Department of Science & Technology, Govt. of India for the partial support of this work.

References

[1] *ARM AMBA Specification Rev 2.0*,http://www.arm.com
[2] Armoni, R. *et. al.* The ForSpec Temporal Logic In *Proc. of TACAS'2001*.
[3] HyperTransport 2.0, http://www.hypertransport.org
[4] Pellauer,M. Mieszko,L. Rishiyur,N "Synthesis of Synchronous Assertions with Guarded Atomic Actions" MEMOCODE,2005.
[5] OpenVera Assertions LRM 2.0. http://www.open-vera.com
[6] PCI Local Bus Specification Revision 2.2 (1998). http://www.pcisig.com/specifications/conventional
[7] Sugar Formal Property Language Reference Manual. http://www.haifa.il.ibm.com/projects/verification/sugar/
[8] FOCs Homepage http://www.haifa.il.ibm.com/projects/verification/focs/
[9] System Verilog. http://www.systemverilog.org/
[10] Synopsis Design Compiler. http://www.synopsys.com/products/logic/design_comp_cs.html
[11] OVL Library. www.eda.org/ovl