Identifying a Subset of SystemVerilog Assertions for Efficient Bounded Model Checking

Robert Wille[1] Görschwin Fey[1,2] Marc Messing[1] Gerhard Angst[3] Lothar Linhard[3] Rolf Drechsler[1]

[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]VLSI Design & Education Center, University of Tokyo, Tokyo 113-0032, Japan
[3]Concept Engineering GmbH, 79111 Freiburg, Germany

{rwille,fey,marcm,drechsle}@informatik.uni-bremen.de
{gerhard.angst,lothar.linhard}@concept.de

Abstract

Integrating design and verification becomes more and more important due to the increasing complexity of today's circuits and systems. SystemVerilog is a description language that embeds verification goals with the help of SystemVerilog Assertions (SVAs). Often SVAs are used in simulation based verification. But in the recent past first applications in formal verification have been considered, too.

In this paper we present an approach to prove SVAs by induction based Bounded Model Checking (BMC). Since checking SVAs is computationally very complex, we define a subset which is sufficient for many practical purposes. For each restriction a rationale is given. The creation of the BMC instance for this subset is explained in detail. Case studies show the application of our approach.

1. Introduction

Steadily increasing complexity of circuit and system design demands a more and more elaborated design flow. Within this flow ensuring functional correctness of a design is one of the bottlenecks. One way to address this problem is the tight integration of design and verification tasks. To this end new description languages embed the verification objectives in the design description.

One such case is assertion based verification [6]. As a concrete example SystemVerilog [7] includes *SystemVerilog Assertions* (SVAs). Often these assertions are used to cross-check the behavior while a design is exercised. But relying on simulation based verification approaches is inherently incomplete. Only a few of the possible scenarios can be considered. In contrast model checking [2] proves the validity of an assertion for any possible situation.

Model checking of SVAs is PSPACE-complete even for a "simple subset" [3]. If more elaborated features, like intersection of regular expressions, the use of local variables, or instantiation of properties are used, the problem becomes EXPSPACE-complete [3]. However, a number of researchers considered synthesis of SVAs into finite state machines [4, 8]. Once an assertion is synthesized model checking can be formulated as a reachability analysis on the product machine of design and assertion. This leads to the above mentioned complexity issues.

Bounded Model Checking (BMC) [1] has been shown to be much more efficient for practical verification problems. BMC reduces the verification problem to a problem of *Boolean Satisfiability* (SAT) and then searches for counterexamples in executions whose length is bounded. Especially induction proofs [11] for safety properties are very efficient [12]. Therefore, proving SVA properties using induction based BMC is considered in this work.

Checking SVAs is computationally very complex in general while for practical purposes a subset is sufficient. In this work we

1. identify a subset of SVAs which guarantees SAT instances of acceptable size,
2. discuss the restrictions (i.e. give reasons why some SVA operators are not supported), and
3. give a detailed description of the translation of SVAs into a BMC instance.

Our approach handles safety properties that argue over a bounded number of time steps. Liveness properties, the use of local variables and infinite sequences are excluded. Furthermore, a single clock signal is assumed.

First, the restrictions on SVAs are introduced and the rationale for each restriction is discussed. For the resulting subset of SVAs the transformation into a SAT instance is explained in detail. This transformation is non-trivial since the respective time information of the sequences has to be determined first. Some case studies at the end of this paper show, that the defined subset can be used to prove useful properties for a given design.

This paper is structured as follows: The basics of BMC and SVAs are revisited in the next section. Here, also the internal representation of SVAs as used in this work is given.

This research work was supported in part by the German Federal Ministry of Education and Research (BMBF) in the Project HERKULES under the contract number 01M3082.

In Section 3 the supported subset and restrictions on SVAs are introduced. The construction of a BMC instance is described in Section 4, i.e. the determination of the time information as well as the encoding of some exemplary operations are introduced. Section 5 shows the application of our approach in terms of some case studies. Finally, conclusions are drawn.

2. Preliminaries

In this work we present the application of BMC to SVA properties. To keep the paper self-contained the basics of BMC and SVAs are given in this section. Afterwards the internal representation of SVAs, which is used in this work, is introduced.

2.1. Bounded Model Checking

In the original formulation, BMC starts from the initial state and searches for counterexamples of increasing length [1]. A property is finally proven if the state space diameter is reached. In contrast induction based BMC can prove the validity of a safety property using a significant smaller problem instance [11]. The proof becomes even more simple if the initial state for a property is not restricted at all. Theoretically this may cause "false negatives": the property fails in an unreachable state. But in practice a verification engineer can usually supply an assumption to define a valid state. The property is then checked under this assumption. Similar approaches have been applied to commercial designs [12]. This type of properties is considered here.

More formally, the design is unrolled for t_{bnd} cycles and the property is attached to the unrolled design. The symbol $f_{\delta}^{t_{bnd}}$ denotes the propositional formula describing the design δ unrolled for t_{bnd} time steps. A second propositional formula f_P describes the property P. This formula is satisfiable iff the property holds under a certain assignment. Thus, the verification problem is reduced to proving the unsatisfiability of $\overline{f}_P \wedge f_{\delta}^{t_{bnd}}$, i.e. there does not exist an input and state assignment to the design such that the property is violated.

Finding a satisfying assignment yields a counterexample, i.e. a trace $x^0 \ldots x^{t_{bnd}}$ of length t_{bnd} that provides values for the primary inputs x of δ at each time step.

2.2. Assertions in SystemVerilog

SystemVerilog provides two kinds of assertions: *Immediate assertions*, which check if an expression in a procedural block is true at a given instance of time, and *concurrent assertions*, which check a behavior of the design over a period of time. Handling immediate assertions is straight forward; therefore we concentrate on concurrent assertions in the following.

A concurrent assertion P – also called SVA property – is checked each time when a clock event `clkevent` as specified in the assertion occurs:

`always assert property (@(clkevent)` P`)`

The occurrence of the clock event also triggers the proceeding of time.

The atoms of concurrent assertions are so called sequences. A sequence R is a regular expression that describes the behavior of signals over time. The semantics is defined with respect to traces. On a given trace a sequence either matches or does not match. All SVA sequences can be described using the abstract syntax

$$\begin{aligned} R ::= \quad & b|(1, v = e)|(R)|(R \;\#\#1\; R)|(R \;\#\#0\; R)| \\ & (R \text{ or } R)|(R \text{ intersect } R)|\textit{first\_match}(R)| \\ & R\;[*0]|R\;[*1:\$], \end{aligned}$$

where R denotes a sequence, b denotes a Boolean expression, v denotes a local variable name, and e denotes an expression.

An SVA property P is composed of sequences, i.e.

$$\begin{aligned} P ::= \quad & R|(P)|\; \textit{not}\; P|(P \textit{ or } P)|(P \textit{ and } P)| \\ & (R\,|\!-\!>P)|\text{disable iff}(b)P. \end{aligned}$$

Additionally, *recursive properties* may be defined using instantiation. The semantics of sequences and properties is explained in more detail in [7].

2.3. Internal Representation of SVAs

In the following we assume that properties are given by a graphical structure similar to the parse tree.

A *terminal* is a signal of the circuit which is considered by an SVA. In the following all terminals of an assertion are stored in the set T.

The set O contains the operations of SVA properties and sequences. Besides the basic operators given in the abstract syntax above, derived operators (e.g. *and* for sequences) are also in O to directly translate them into the BMC instance.

An *SVA property* is represented by a *Directed Acyclic Graph* (DAG) $G = (V, E)$. For each operation of the assertion (i.e. each subexpression) a vertex $v \in V$ exists, which has one or two predecessors $\textit{first}(v), \textit{second}(v) \in V$ (the *operands*). The leaves of the DAG represent the terminals of the assertion.

The function $\textit{operation} : V \to O \cup T$ maps each vertex $v \in V$ to its respective operation $\textit{op} \in O$ or – if v is a leaf – to its respective terminal $t \in T$.

3. Supported Subset of SVAs

The aim of this work is to apply induction based BMC to SVAs within acceptable run times and moderate memory requirements. To this end a subset of SVAs is defined and some further restrictions are applied:

- Reachability analysis is not applied. Reachability analysis is typically very expensive, while often a valid state with respect to a property can be described by a designer.

- The problem instance to check a property must be within an acceptable size. For that reason the length of traces that have to be considered is restricted to a fixed upper bound that does not depend on the design but only on the property. This restriction is acceptable for most designs since they typically respond to requests within a bounded time.

The case study in Section 5 shows, that the supported subset of SVAs is powerful enough to define useful properties. In the following, the restrictions applied to the different layers of SVAs are discussed.

3.1. Clock Events

The clock event that triggers a property might be a complex description. To decide whether a clock event occurs, corresponds to checking a *Linear Temporal Logic* (LTL) [10] property of the form $GF(\texttt{clkevent})$. On the other hand a circuit design often has an intrinsic notion of time defined by a dedicated clock signal clk. Thinking in terms of properties related to clk-ticks is natural to the designer and verification engineer.

Therefore we assume a single clock signal clk for the design under verification and that all assertions and registers are triggered by the rising edge of this signal, i.e. the property has the form
`always assert property(@(posedge` clk`)`P`)`.

As a result the occurrence of the clock event does not have to be checked, but is guaranteed. Assuming this restriction, we remove the clock event from the description of a property in the following and simply write P.

3.2. Sequence Layer

Most operators of the sequence layer are supported. Exceptions are the $[*1:\$]$-operator, the *first_match*-operator and local variables.

The operator $[*1:\$]$ describes a sequence with an unspecified upper bound. To decide whether all traces of a design match (or do not match) such a sequence, the design may have be unrolled to up to its sequential depth[1]. As a result the size of the problem instance depends on the design and may be proportional to the number of states of the design. Therefore the operator $[*1:\$]$ is currently not supported. This guarantees that a sequence argues over a bounded number t_{bnd} of time steps.

The *first_match*-operator triggers a property only once after the initial state. Similar to the previous consideration this first match depends on the design: to decide that the match never occurs, the design has to be unrolled up to its sequential depth. Therefore this operator is also excluded from the supported subset.

Moreover, local variables are excluded from consideration. It has been proven that supporting local variables makes the model checking problem EXPSPACE-complete [3] in general. One approach which addresses this problem has been introduced in [8]. Investigating whether local variables can be supported within our restricted subset is left for future work.

This leads to the following syntax for the supported subset of sequences:

$$\begin{aligned} R ::= \quad & b|(R)|(R\ \#\#1\ R)|(R\ \#\#0\ R)|(R \text{ or } R)| \\ & (R \text{ intersect } R)|R\ [*0]|R \end{aligned}$$

3.3. Property Layer

Further restrictions apply to the property layer. The individual operators and whether they can be supported safely is discussed in the following.

Assume that for a given sequence R a Boolean formula f_R is available that is satisfiable iff the design may produce a trace $x^0 \dots x^{t_{bnd}}$ that matches R (the creation of f_R is explained in Section 4). Now, given an SVA property P, a Boolean formula f_P is created that is unsatisfiable iff the property holds in all states and under all traces. Iff the underlying SAT problem is satisfiable, the property fails and a counterexample can be derived from the solution.

The most simple property is a sequence: $P = R$. A counterexample is a trace of the design where R does not match. This property is encoded as the formula $f_P = \overline{f}_R \wedge f_\delta^{t_{bnd}}$, i.e. the design is unrolled for t_{bnd} time steps and the sequence is not matched. Thus, a property consisting of a single sequence is supported, if the sequence conforms to the subset defined above.

Given a property P', a new property $P = \textit{not}\ P'$ can be derived. Assume, for any given trace of length t_{bnd} the validity of P' can be decided using $f_{P'}$. Thus the validity of P on any trace is known. Therefore the formula $f_P = f_{P'} \wedge f_\delta^{t_{bnd}}$ is satisfiable, iff there exists a trace of length t_{bnd} such that P does not hold. Note, that here the bounded length is necessary to decide whether P holds or not. This guarantees that this decision is never pending due to future obligations formulated in the property.

Composed properties using the operators *or* and *and* can be handled similarly, if both operands are supported.

An implication $P = R\ |->\ P'$ holds iff P' holds each time after R matches, i.e. the following formula is satisfiable, iff P does not hold: $f_P = (f_R \wedge \overline{f}_{P'}) \wedge f_\delta^{t_{bnd}}$. Thus, implications are supported, if P' and R can be handled.

The operator *disable_iff* disables checking a property after a certain condition has been met. This operator is not supported for similar reasons as the *first_match*-operator.

This leads to the following syntax for the supported subset of SVA properties:

$$P ::= R|(P)|\text{not } R|(P \text{ or } P)|(P \text{ and } P)|(R|->P)$$

Recursive properties are not considered as they may be of unbounded length as the following example shows:

```
property P(a);
a and (1'b1 |=> P(a));
endproperty
```

The previous observations rely on the assumption that a sequence R can be translated into a formula f_R. This step is explained in the next section.

[1]The sequential depth is the longest trace where any state only occurs once.

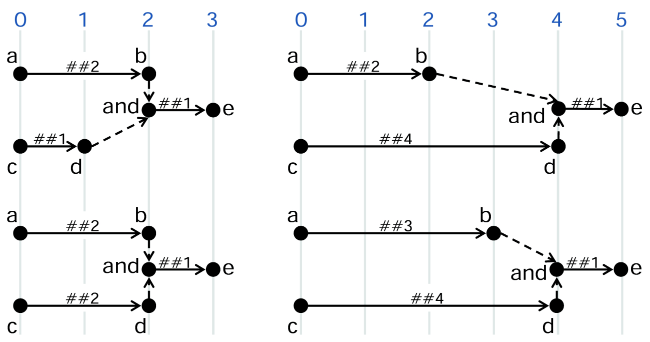

Figure 1. Different possibilities

4. Matching Sequences

Sequences can be seen as the atoms of SVAs. Therefore we show how to decide whether an unrolled design may produce a trace that matches a given sequence or not. The sequences considered here are of bounded length t_{bnd}. This length is determined in a first stage. Moreover, for each sequence the earliest time step for the evaluation and the latest time step are determined. Based on this information, the sequence is unrolled in the second stage. Before explaining the algorithm, a motivating example shows some of the difficulties during this process.

4.1. Motivating Example

Attaching a simple sequence to the appropriate signals in the unrolled design is straight forward. For example, the sequence $a\ \#\#1\ b$ refers to signal a in the initial time step and to b in the next time step. If multiple sequences are joined by operators, the referenced time step depends on the evaluation of previous operations of the sequence and thus is not fixed. Consider the sequence $(R_1\ and\ R_2)\ \#\#1\ R_3)$[2]. By definition, R_3 is evaluated one time step after the first part ($R_1\ and\ R_2$) is matched. The first part is matched after R_1 and R_2 are matched. As a result the time step to start matching R_3 depends on the evaluation of the first part.

Example 1 *Consider the sequence*[3]

$$((a\ \#\#[2..3]\ b)\ and\ (c\ \#\#[1..4]\ d))\ \#\#1\ e.$$

The time steps for the consideration of signals a, b, c and d are directly given. To match e, first of all the result of the and-operation has to be known. Thus, the sequence above may be described by the formula[4]

$$(((a_0 \wedge b_2) \vee (a_0 \wedge b_3)) \wedge ((c_0 \wedge d_1) \vee \cdots \vee (c_0 \wedge d_4))) \wedge e_{1+x},$$

where the concrete time step, in which signal e is considered, is unknown (denoted by e_{1+x}).

[2]The sequence-operator and is a derived operator [7].

[3]The *interval operator* $\#\#[a..b]$ is derived from the basic operators $\#\#1$ and or.

[4]A signal is considered in the time step denoted by the index.

Figure 1 shows four of the possible combinations (the shortest and the longest one) which match the complete sequence. Signal e has to be considered at time step 3 at the earliest and at time step 5 at the latest. Furthermore, the whole sequence holds e.g. at time step 3 only if signal a and c hold in time step 0, signal b holds in time step 2 and signal d holds in time step 1 or 2, respectively. That is, the sequence is unrolled to the following propositional formula:

$$\begin{aligned} & (((a_0 \wedge b_2) \wedge ((c_0 \wedge d_1) \vee (c_0 \wedge d_2))) \wedge e_3) \\ \vee\ & ((((a_0 \wedge b_2) \vee (a_0 \wedge b_3)) \\ & \wedge((c_0 \wedge d_1) \vee (c_0 \wedge d_2) \vee (c_0 \wedge d_3))) \wedge e_4) \\ \vee\ & ((((a_0 \wedge b_2) \vee (a_0 \wedge b_3)) \wedge (c_0 \wedge d_4)) \wedge e_5) \end{aligned}$$

Thus, the time steps at which signals (or operations of a sequence) have to be considered are not fixed but depend on the evaluation of previous operations. Since concrete time steps are needed to unroll a property, these *time spans* have to be calculated.

4.2. Determining Time Information

As shown in Example 1, each operator and terminal have to be considered at multiple time steps. For this purpose we use the following definition:

Definition 1 *The* time span*, during which an operation or a terminal of a sequence has to be considered, is denoted by an interval I, i.e. a closed bounded set of positive integers $I = [l..h] = \{x \mid l \leq x \leq h\}$. Furthermore, the* maximum *of two intervals I_1 and I_2 is defined by*

$$\max(I_1, I_2) = [\max(l_1, l_2)..\max(h_1, h_2)].$$

The sum *of two intervals I_1 and I_2 is defined by*

$$I_1 + I_2 = [(l_1 + l_2)..(h_1 + h_2)].$$

Based on the internal representation of SVA properties (see Section 2.3) and the previous definition we provide an algorithm that

1. calculates the time spans for each subsequence and
2. unrolls the sequence for the formal proof.

The calculation of the time spans is done by a depth-first search over all operations of the sequence, i.e. over all vertices of the DAG starting from the root vertex. For each operation (i.e. for each $v \in V$) a function *calcTimeInfo* is recursively called. The pseudo-code of this function is shown in Figure 2. The algorithm starts at the root vertex with the argument $I = [0..0]$. Then, the calculation proceeds in four steps for the current vertex:

(a) Calculating the time span I_{1st} for the first operand (lines 4-8),

(b) if necessary, modifying I depending on the operation (lines 10-15),

```
(1)  calcTimeInfo ($I$)
(2)    // $I$ is the time information from already
       considered operations or initial $[0..0]$
(3)    – Step (a) ——————————————
(4)    $pred = first(v)$;
(5)    if ($operation(pred) \in T$)
(6)      $I_{1st} = I$;
(7)    else
(8)      $I_{1st} = pred$.calcTimeInfo($I$);
(9)    – Step (b) ——————————————
(10)   if ($operation(v) == int$)
(11)     $I = I_{1st} + I_{int}$;
(12)   if ($operation(v) == \ |=>$)
(13)     $I = I_{1st} + [1..1]$;
(14)   else if (...)
(15)     // similar for other operations
(16)   – Step (c) ——————————————
(17)   $pred = second(v)$;
(18)   if ($operation(pred) \in T$)
(19)     $I_{2nd} = I$;
(20)   else
(21)     $I_{2nd} = pred$.calcTimeInfo($I$);
(22)   – Step (d) ——————————————
(23)   if ($operation(v) == and$)
(24)     $I_v = max(I_{1st}, I_{2nd})$;
(25)   else if ($operation(v) == int$)
(26)     $I_v = I_{2nd}$;
(27)   else if (...)
(28)     // similar for other operations
(29)   return $I_v$;
```

Figure 2. Determining time information for vertex $v \in V$

(c) calculating the time span I_{2nd} for the second operand (lines 17-21), and

(d) determining the time span I_v for the current operation (lines 23-28).

Steps (a) and (c) are done by recursive calls of *calcTimeInfo*. First, the respective operand is acquired (line 4/17). If this operand is a terminal of the sequence (i.e. a signal of the circuit), the current value of I is still the time span in which this operand (i.e. this signal) becomes relevant (line 6/19). Otherwise, the operand is another operation and *calcTimeInfo* is recursively called to determine the time span of this operation (line 8/21).

In some cases the interval I has to be adjusted before the second operand is considered. This is done in step (b). Depending on the current operation the value of I is changed (lines 10-15). For example if an SVA interval operator $[a : b]$ is considered, the interval $I_{int} = [\#\#a..b]$ of this sequence is added to I_{1st} resulting in the time span, in which the second operand starts to evaluate. Note, for some operations (e.g. and) this modification is not necessary, since both operands are evaluated independently of each other.

Finally, in step (d) the time span I_v for the current operation is determined and returned. Again, this depends on the type of the operation. For example, sequences are evaluated at the same time step as the second operand (i.e. $I_v = I_{2nd}$). In contrast e.g. for an and-operation both operands can be within independent time spans. Since both operands have to be evaluated to perform an and, the maximum of both intervals is calculated (i.e. $I_v = max(I_{1st}, I_{2nd})$).

Example 2 *The following sequence is used to illustrate the algorithm in more detail:*

$$f \#\#[1..2] (((a \#\#[2..3]\, b)\ and\ (c \#\#[1..4]\, d)) \#\#1\, e)$$

Figure 3 shows the resulting time spans for this sequence. Each operation (vertex) of the sequence is represented by a box. Within the boxes the calculated time spans I_{1st} and I_{2nd} for the operands (first and second row) as well as I_v for the operation itself (last row) are shown. Furthermore, the respective parameter I and the return values of the recursive calls (equal to I_v of the last operation) are annotated at the respective edges. As mentioned above each depth-first search starts with the initial interval $I = [0..0]$.

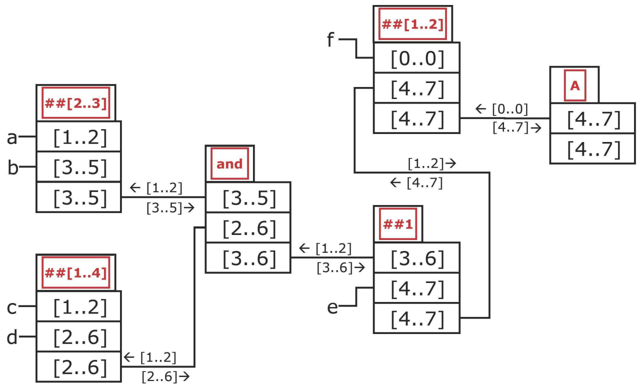

Figure 3. Example for time information

The sequence ##[1..2] is considered first. Here, the first operand is a terminal of the sequence. Thus, this operand becomes relevant at time step 0 only ($I_{1st} = I = [0..0]$). Since the interval has to be considered, the second operand (and all subordinate operations) matches one or two time steps later. Thus, calcTimeInfo is recursively called with $I = [0..0] + [1..2] = [1..2]$ for the second operand. The base case of this recursion is reached at operation ##[2..3]. Here, both operands are terminals resulting in $I_{1st} = I$ and $I_{2nd} = I$, while between these assignments I is modified because of the interval (i.e. $I = [1..2] + [2..3] = [3..5]$). This also applies to the interval ##[1..4] which is considered next (because of the and-operation this is done independently from interval ##[2..3]). In doing so, for both operands of the and-operation the time span is known ($I_{1st} = [3..5]$ and $I_{2nd} = [2..6]$). Since both operands have to be considered *to perform the* and*, the earliest time step, in which this can be done is* $max(3,2) = 3$*; the latest is* $max(5,6) = 6$*. Thus, the* and*-operation is evaluated within the time span* $I_v = [3..6]$*. In the same way, the time spans of the remaining operations are calculated.*

Overall, when all recursive calls of *calcTimeInfo* terminate the following information is available: (1) the number of time frames t_{bnd} the sequence needs to evaluate and (2) for each vertex v the time span I_v within the subsequence rooted at v has to be evaluated. While (1) provides the number of time steps the design has to be unrolled, the time spans are needed to unroll the sequence.

4.3. Unrolling Sequences

If the time spans of all operations are known, the sequence can be encoded as a Boolean formula f_R. In the following an abstract description of this transformation is given for the whole formula at first. A more detailed description for some concrete operations follows.

The subsequence rooted at a vertex $v \in V$ has to be evaluated for each time step $i \in I_v$. The corresponding constraints are denoted by c_i^v. Whether all constraints in c_i^v are met, is stored by an auxiliary variable a_i^v:

$$c_i^v \Leftrightarrow a_i^v$$

Remark 1 *Since a terminal (represented by* $v \in V$*) can be encoded as a single variable the auxiliary variable* a_i^v *can be omitted and instead* c_i^v *can be used directly. However, to simplify the notation* a_i^v *is used to represent terminals as well.*

Using this notation, the conjunction of all constraints $a_i^v \Leftrightarrow c_i^v$ encodes the sequence (i.e. the whole formula f_R):

$$f_R = \bigwedge_{v \in V} \bigwedge_{i=l}^{h} a_i^v \Leftrightarrow c_i^v, \text{ with } I_v = [l..h]$$

Example 3 *Again, the sequence defined in Section 4.2 is considered. All the operations are encoded with respect to their time span as constraints* $c_3^{\#\#[2:3]}$*, ...,* $c_5^{\#\#[2:3]}$*,*c_3^{and}*, ...,* c_6^{and} *and so on. The Boolean formula* f_R *representing the whole sequence is:*

$$f_R = a_3^{\#\#[2:3]} \Leftrightarrow c_3^{\#\#[2:3]} \wedge \cdots \wedge a_5^{\#\#[2:3]} \Leftrightarrow c_5^{\#\#[2:3]}$$

$$\wedge a_3^{and} \Leftrightarrow c_3^{and} \wedge \cdots \wedge a_6^{and} \Leftrightarrow c_6^{and} \wedge \ldots$$

Now, the encoding for two selected operations is described in more detail. Here, the encoding of the respective constraints c_i^v depends on the type of the operation (i.e. on $operation(v)$) and on the time step i to be represented.

First, the interval operation (represented by v_{int}) is considered. Here, the interval (e.g. $\#\#[2..3]$) may not be applicable in all time steps since this contradicts the determined intervals of the operands (i.e. the time steps when operands match). The following example illustrates the problem.

Example 4 *The constraints to encode the operation* $a \ \#\#[2..3] \ b$ *of the sequence defined in Section 4.2 are created. The second operand (signal* b*) is considered in the same time step as the whole subsequence matches. The first operand (signal* a*) is considered two or three time steps earlier. Thus, when considering time step* 3*, one may expect that signal* a *is considered within time steps* $[0..1]$*. But since this contradicts* $I_a = [1..2]$*, signal* a *is considered at time step* 1 *only.*

Thus, the operation $A \ \#\#[l..h] \ B$ represented by v_{int} with A (B) is represented by $a = first(v)$ ($b = second(v)$) and $I_a = [l_a..h_a]$ is encoded for time step $i \in I_{v_{int}}$ by $c_i^{v_{int}}$ as follows:

$$a_i^b \wedge (\bigvee_{j=max(i-h,l_a)}^{min(i-l,h_a)} a_j^a)$$

As a second example, the and-operation (represented by v_{and}) is considered. Here, both operands are evaluated starting from the same initial state but may argue over different time spans. However, the and-operation does not match until both operands match. Thus, if this operation is evaluated at a time step $i \in I_{v_{and}}$ one operand has to be evaluated at this time step, too, while the other has to be evaluated at the same or an earlier time step. Since again, the determined time spans I_{1st} and I_{2nd} of the respective operands have to be considered, this results in three different cases: The first operand evaluates at a time step greater than the higher bound of the time span of the other operand or vice versa or both operands evaluate at the same time step.

More formally, consider the operation $A \ and \ B$ represented by vertex v_{and}. A is represented by vertex $a = first(v)$ with $I_a = [l_a..h_a]$ and B is represented by vertex $b = second(v)$ with $I_b = [l_b..h_b]$. Then, the constraint $c_i^{v_{and}}$ encodes $A \ and \ B$ for time step $i \in I_{v_{and}}$ as follows:

$$\begin{array}{ll}
a_i^a \wedge (a_{h_b}^b \vee \cdots \vee a_{l_b}^b) & \text{, if } h_b < i \leq h_a \\
a_i^b \wedge (a_{h_a}^a \vee \cdots \vee a_{l_a}^a) & \text{, if } h_A < i \leq h_b \\
(a_i^a \wedge (a_{i-1}^b \vee \cdots \vee a_{l_b}^b)) & \\
\vee (a_i^b \wedge (a_{i-1}^a \vee \cdots \vee a_{l_a}^a)) & \text{, if } i \leq h_a, h_b
\end{array}$$

Example 5 *The constraint representing the operation* $op_{and} = (a \ \#\#[2..3] \ b) \ and \ (c \ \#\#[1..4] \ d)$ *in the 6th time step of the sequence defined in Section 4.2 is:*

$$c_6^{op_{and}} \Leftrightarrow v_6^{op_{\#\#[1..4]}} \wedge (v_5^{op_{\#\#[2..3]}} \vee \cdots \vee v_3^{op_{\#\#[2..3]}})$$

Besides these two encodings, the constraints c_i^v for all other operations (represented by $v \in V$) are encoded in a similar way[5]. In total no more than $O(|V| \cdot t_{bnd})$ variables are needed to encode the SystemVerilog sequence as a propositional formula (i.e. linear in the number of operations and the maximal number t_{bnd} of time steps). Moreover, in many cases this number is smaller, since many operations are only considered over a smaller time span than $[0..t_{bnd}]$.

[5]Due due page limitation we will not describe the remaining ones.

Using this transformation of a sequence R into a propositional formula f_R and the time information to determine t_{bnd}, a property P can be verified as explained in Section 3. Some case studies using this encoding for proving SVA properties are given in the next section.

5. Case Studies

The proposed algorithms have been implemented in C++. MiniSat [5] is used as the underlying SAT solver. In the following we show in some case studies how the defined subset can be used to prove useful properties for a given design. Therefore, we consider a scalable arbiter in more detail. Furthermore, we provide further results for a counter and an ALU. All experiments have been carried out on an AMD64 4200+ with 2GB of main memory.

5.1. Arbiter

We tested our approach with an arbiter as described in [9]. The arbiter is a scalable design that handles the access of n clients to a resource. Usually priority scheduling is used to serve the clients. In the case that there are too many requests the arbiter switches to a round robin scheme. This guarantees that all clients can access the resource after at most $2n$ time steps. The arbiter is implemented by a composition of n cells. Cell i ($1 \leq i \leq n$) is connected to the client with id i by a request input $\texttt{req}_i$ and an acknowledge output $\texttt{ack}_i$. The client with the lowest id has the highest priority. If a client signals a request, the client waits for at most $2n$ clock cycles before it is served regardless of other client's requests.

Two properties are discussed in the following: (1) A safety property, guaranteeing that no two clients access the resource at the same time, and (2) a liveness property, proving the access to the resource within $2n$ clock cycles.

The SVA code for the safety property (1) and $n = 3$ is composed of several parts:

- Assumption – only a single token is present:
```
property sumToken;
(t1 + t2 + t3) == 1;
endproperty;
```
- Proof objective – at most one `ack`-signal is 1:
```
property sumAck;
(ack1 + ack2 + ack3) <= 1;
endproperty;
```
- Complete property:
```
assert property (@ (posedge clk)
not (sumToken ) or sumAck);
```

For this property one time step is considered. The size of the problem instance is proportional to the number of clients n.

To check the liveness property (2), the property has to be formulated as a safety property, saying that client i is always served within $2n$ clock cycles, if it keeps the req_i until ack_i is granted. Again, the property is composed of several parts. The SVA code for $n = 3$ and a client with id 3 is:

- Assumption – `sumToken` (as above).
- Assumption – client 3 keeps $\texttt{req}_3$ until granted $\texttt{ack}_3$:
```
property keepReq;
req3 and
( !ack3 |=> req3) and
( !ack3 ##1 !ack3 |=> req3) and
...
( !ack3 ##1 ... ##1 !ack3 |=> req3);
endproperty;
```
- Proof objective – within $2n$ cycles $\texttt{ack}_3$ is granted:
```
property ackWithin2n;
not ( !ack3 ##1 !ack3 ... ##1 !ack3);
endproperty;
```
- Complete property:
```
assert property (@ (posedge clk)
not (sumToken and keepReq) or
ackWithin2n);
```

To check these properties, both the SVA as well as the design have been encoded into a BMC instance as described in Section 4. For different values of n, the size of the respective instances (in terms of number of variables) and the run time needed to solve the properties are given in Figure 4 and Figure 5, respectively.

As expected, the size of the instance for the safety property (1) is proportional to the number of clients. In contrast, the liveness property (2) has quadratic size. Here, not only the design grows with increasing n but also the number of considered time steps (i.e. $2n$).

Regarding run time, similar effects can be observed. Proving the second property (arguing over $2n$ clock cycles) takes significantly more run time than proving the first one (arguing over one time step)[6]. However, both properties can be checked efficiently. Further experimental evaluation is documented in the next section.

5.2. Counter and ALU

The proposed approach has been applied to two more designs: a counter and a simple *Arithmetic Logic Unit* (ALU). The counter is a finite state machine with eight states representing binary encodings of the natural numbers from 0 to 7. The finite state machine steps through the states in increasing order and wraps around after state 7. The ALU performs a set of operations as e.g. addition, multiplication and division. On these circuits a set of (failing and holding) properties – given in SVAs – are checked.

The results are summarized in Table 1. Column ASSERTION lists the properties with the result (holds or fails). The number of time steps t_{bnd} the design has been unrolled are given in column t_{bnd}. The following two columns give the number of variables and the number of clauses of the SAT instances, respectively. Finally, the run time of the BMC check (in CPU seconds) is given in the last column. All checks are performed within a short run time.

[6]Note the logarithmic scale of the y-axis.

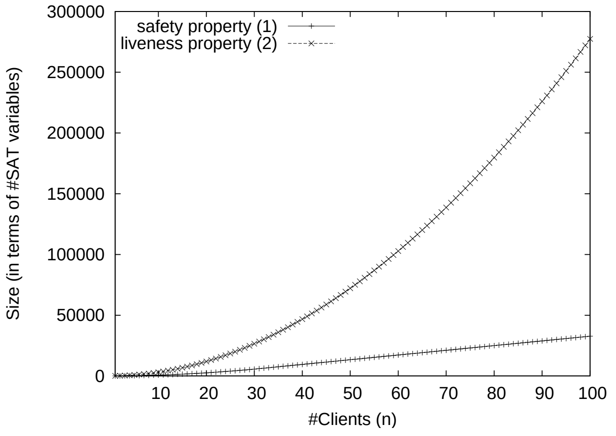

Figure 4. Size of the SAT instances

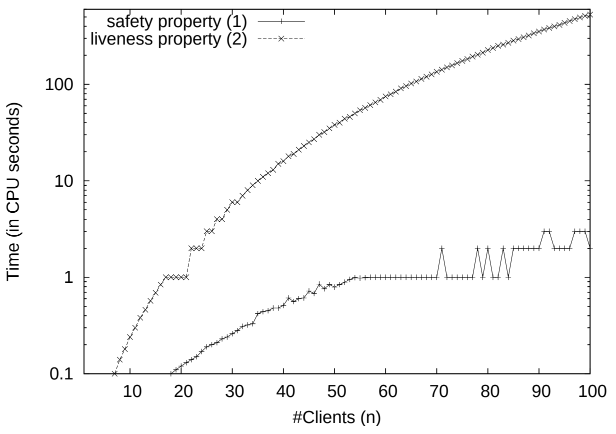

Figure 5. Run-times to check SVAs

6. Conclusion

In this paper we presented an approach to prove SVA properties using induction based BMC. To handle the complexity we defined a subset, which is powerful enough for practical purposes. The reasons for restricting SVAs have been discussed in detail. Furthermore, the algorithm which handles SVA sequences with different matching times has been explained. Experiments on different designs show, that the restricted subset is powerful enough to describe useful properties that are efficiently checked by our approach.

Table 1. Further experimental results

ASSERTION	t_{bnd}	#VARS	#CLSES	TIME
COUNTER				
p1 (holds)	2	229	377	0.02s
p1 (fails)	2	229	377	0.02s
p2 (holds)	11	1675	2781	0.07s
p2 (fails)	11	1675	2781	0.16s
p3 (holds)	11	2096	3574	0.11s
p3 (fails)	11	2233	3803	0.28s
p4 (holds)	9	2408	3938	0.15s
p4 (fails)	9	120	2248	0.19s
p5 (holds)	11	2501	4238	0.15s
p5 (fails)	11	2501	4238	0.20s
ALU				
p1 (holds)	2	1427	2724	0.28s
p1 (fails)	2	1427	2724	0.22s
p2 (holds)	9	9893	19306	0.53s
p2 (fails)	8	8626	16773	0.65s
p3 (holds)	4	349	94	0.05s
p3 (fails)	4	2990	5447	0.30s
p4 (holds)	2	1166	2125	0.15s
p4 (fails)	2	1166	2125	0.20s

References

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.

[2] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conf.*, pages 46–51, 1990.

[3] D. Bustan and J. Havlicek. Some complexity results for systemverilog assertions. In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 205–218, 2006.

[4] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti. Synthesis of system verilog assertions. In *Design, Automation and Test in Europe*, pages 70–75, 2006.

[5] N. Eén and N. Sörensson. An extensible SAT solver. In *SAT 2003*, volume 2919 of *LNCS*, pages 502–518, 2004.

[6] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, 2003.

[7] IEEE System Verilog Working Group. *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification (IEEE Std 1800-2005)*. IEEE, 2005.

[8] J. Long and A. Seawright. Synthesizing SVA local variables for formal verification. In *Design Automation Conf.*, pages 75–80, 2007.

[9] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[10] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[11] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Int'l Conf. on Formal Methods in CAD*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

[12] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. Cost-efficient block verification for a UMTS up-link chip-rate coprocessor. In *Design, Automation and Test in Europe*, volume 1, pages 162–167, 2004.