Improving Simulation-Based Verification by Means of Formal Methods*

Görschwin Fey Rolf Drechsler
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
e-mail: {fey,drechsle}@informatik.uni-bremen.de

Abstract— The design of complex systems is largely ruled by the time needed for verification. Even though formal methods can provide higher reliability, in practice often simulation based verification is used. Large testbenches are created and if the design produces the correct output for all stimuli it is said to be correct. But there is no guarantee that the testbench is complete in the sense that it contains test-cases for all "important" situations.
We propose an approach to detect "gaps" in testbenches, i.e. behavior that is not tested. The approach relies on automatic generation of properties from the testbench in terms of a formal property language. By construction the properties are valid within the testbench. A model checker proves the validity of the property on the design. If this proof succeeds, the testbench covers all possible situations for given signals. In case of failure counter-examples are produced. These counter-examples represent behavior that is not tested, i.e. a gap in the testbench. The feasibility of the approach is underlined by experiments.

I. Introduction

Verification is a major issue in the design of integrated circuits and systems. According to Moore's law the number of elements in a manufacturable circuit doubles every 18 months. But the design productivity increases at a lower rate. Resulting is a design and verification gap. On the other hand verification of circuits is becoming even more important as circuits are applied in a variety of systems concerned with security issues. Even the use of fast simulators for simulation of several million clock cycles can not ensure functional correctness. Only a fraction of the state space of a design can be explored by simulation, and by this is far from being completely covered. Also applying coverage metrics (e.g. statement or state coverage) and achieving 100% coverage with respect to a certain metric by simulation can not guarantee correctness.

Formal verification methods on the opposite guarantee completeness under any input sequence and in any state of the design. The compliance of a design with the specification is formally verified by model checking [7]. The concept of *Bounded Model Checking* (BMC) [1] and the improvements on engines to proof properties, i.e. SAT-solvers [6, 8], allow to formally verify large parts of a design. Nonetheless, due to familiarity of designers with simulation and the fact that whole systems can not be handled by property checking due to the complexity, simulation is still widely used to check the correctness of a system. For this, large testbenches are created.

Techniques to gather information about the reliability of the verification mostly rely on coverage metrics. Simulation based approaches use monitors during simulation to determine the amount of coverage. Formal approaches are defined for model checking, i.e. they give a notion of coverage achieved by a given set of properties (e.g. [4, 5]).

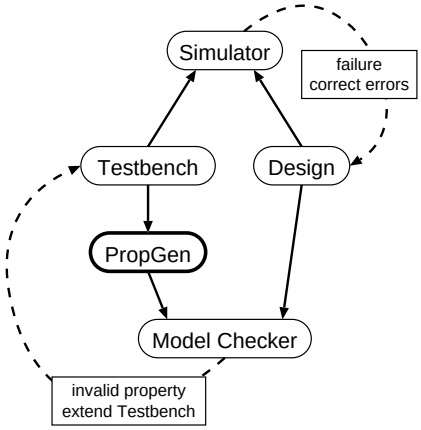


Fig. 1. Integration in the verification flow

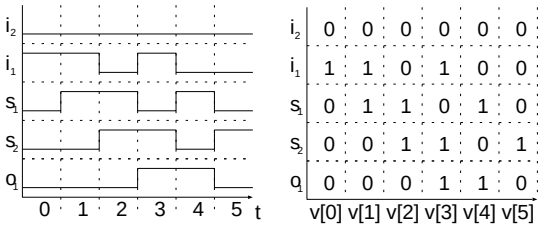
Here, we propose a method to formally analyze a testbench and to check which parts of the functional behavior of a design are not tested. This is done by automatically generating properties in terms of a formal property language from a given simulation trace. Figure 1 shows how this approach is facilitated in the simulation based verification flow. *PropGen* is our tool to generate properties from the testbench. While usually only the simulator is available to check the design by means of the testbench, our tool is employed to analyze the testbench. An invalid property leads to a counter-example produced by the model checker. This counter-example exhibits the behavior that is not examined by the testbench, i.e. a gap in the set of stimuli provided by the testbench. This knowledge can be used, e.g. to extend the testbench. The integration of PropGen and the model checker results in an easy to use push-button tool for analyzing a testbench. The user does not have to know anything about the underlying formal techniques. In summary a crosscheck of testbench and design is established by our method.

Additionally a mechanism to generate more focused properties is provided. The generation of properties showing all dependencies between a certain signals can lead to properties that are too general. By applying restrictions PropGen can be guided to find properties for certain situations, e.g. a particular operation mode of the design.

As a side effect the method provides insight into the design for a verification engineer. Instead of going through the code for a portion of the design, the designer can select some signals and extract their relation automatically from the testbench. Afterwards the engineer has to decide if the property should be valid for the design or not. Then, by means of a property checker the correctness of the design can be checked.

Experimental results show the efficiency of our ap-

*This work was supported in part by DFG grant DR 287/8-1.



(a) Waveform

(b) Vector representation of the waveforms

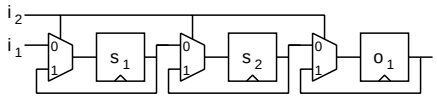


Fig. 2. 1-bit-shift-register

proach: Even generating a property from traces of more than 1 million clock cycles took at most 6 minutes, but usually less than 10 seconds.

II. Preliminaries

This section introduces the necessary basic notions to make the paper self-contained. These are circuits, traces, the type of properties considered and in short the concept of BMC.

A *circuit* has n primary inputs $(i_1 \dots i_n)$, m state bits $(s_1 \dots s_m$, e.g. flip-flops) and p primary outputs $(o_1 \dots o_p)$. A value of an input, output or state bit at time t is indicated by $[t]$, e.g. the value of input i_j at time t is indicated by $i_j[t]$. The values of outputs and state bits are determined by Boolean functions that depend on the values of inputs and state bits at the previous time step.

A *simulation trace* of t_{cyc} clock cycles is denoted by an array of vectors $v^1, \dots, v^{t_{cyc}}$. Each $v[t]$ records the values of inputs, state bits and outputs at time t:

$$v[t] = (i_1[t], \dots, i_n[t], s_1[t], \dots s_m[t], o_1[t], \dots o_p[t])$$

For example consider the waveforms in Figure 2(a). This can directly be mapped into the vector notion that is shown in Figure 2(b). Thus, necessary data can be generated from any simulation trace e.g. the widely used VCD-format.

Note, that internal signals can not be considered using this notation, but the extension is straightforward. Further note, that by this simulation method outputs are modeled as flip-flops. In the following *signal* refers to an input, output or state bit.

In this work a *property* for a circuit is considered to be a propositional formula. Variables are the inputs, state bits and outputs of the circuit associated to a certain time step. The *length of the window* for a property is given by the largest time step referenced by any variable plus one (the first time step is considered to be zero). The property is shifted to an arbitrary time step t by adding t to each time reference. A property is valid for a circuit, if it holds for any simulation trace at any time step for this circuit.

Example 1. *Consider the circuit in Figure 2. This is a 1-bit-shift-register with 2 state registers s_1, s_2 and a registered output o_1. The shift-register has two modes of operation: keep the current value ($i_2 = 1$) and shifting ($i_2 = 0$). During shifting the value of input i_1 is shifted into the register. Therefore after three clock cycles the value is observed at the output o_1.*

This behavior is described by the property "If i_2 is zero on three consecutive time steps, the value of i_1 in the first time step equals o_1 in the fourth time step" which can be written as a formula:

$$\overline{i}_2[t] \cdot \overline{i}_2[t+1] \cdot \overline{i}_2[t+2] \rightarrow (i_1[t] = o_1[t+3]) \qquad (1)$$

The length of the window for this property is 4.

This notion of a property is also used by industrial model checking tools (e.g. [2]). Having a window for the property is not a restriction in practice. Very often the length of the window corresponds to a particular number of cycles needed for an operation in the design. In case of the shift-register this is the number of cycles needed to bring an input value to the output. For a sophisticated design like a processor this can be the depth of the pipeline, i.e. the number of cycles to process an instruction.

Bounded Model Checking (BMC) allows to reduce the sequential problem of proving a property to a combinational one (for details see e.g. [1]). Given a property with a window of length l the design is unrolled l times: l copies are connected in sequential order. Next state bits of a previous copy become state bits for the next copy. The property is proven on the resulting combinational circuit.

III. Generating Properties

The generation of properties is based upon pattern search in a simulation trace. A particular pattern in the trace shows a relationship between signals and by this indicates an underlying property. By taking into account all patterns that occurred the property is generated.

This section introduces the basic procedure for PropGen, how "useful" properties are chosen and how the generation of properties can be guided.

The basic procedure is given in Figure 3: Given is a tuple of signals I and a maximal window t_{max} for the properties to be generated as well as a simulation trace v. In the property a particular time step in the window is assigned to each signal; this time step is not given in advance. An assignment of time steps to signals is called *time relation* in the following. The iteration of all possible time relations T is the outer loop (line 0). At the beginning nothing is known about the property, it is initialized to the constant function 0. Then, at each time step of the trace the behavior of the signals is determined in terms of a pattern (line 3) and included in the property (line 4). The property for a particular time relation T is valid within the trace by construction, because all occurring patterns are considered.

A pattern is the vector that gives the values of signals at the time steps determined by the time relation. The

```
PropGen(I, t_max, v[1], ..., v[t_cyc])
(0)   foreach time relation T
(1)     p(T) = 0
(2)     foreach time step 1 ≤ t < t_cyc
(3)       pat= getPattern(I,T,v,t);
(4)       addPattern(p(T), pat);
(5)     end
(6)   end
```

Fig. 3. Sketch of the property generation

time relation T assigns to a signal $sig \in I$ the time offset within the property $T(sig)$. For a window starting at time t the value inserted for signal sig is $sig[t + T(sig)]$, thus the pattern is determined by the trace. This assignment of values for a pattern is done by *getPattern*. Then, the behavior reflected by the pattern is included in the property by *addPattern*. This is achieved by rewriting the pattern as a conjunction of literals of the variables in I at the time steps determined by T. For a value of 0 in the pattern the negative literal is used, for the value 1 the positive literal is used. This cube determines one valid assignment to the signals, the sum of all these cubes leads to the property $p(T)$.

Example 2. *Consider the trace given in Figures 2(a) and 2(b). Let the tuple of signals I be (i_2, i_1, s_1). And let $T(i_2) = T(i_1) = 0$ and $T(s_1) = 1$. Now, for each time step t the pattern is given by $(i_2[t], i_1[t], s_1[t+1])$. At time steps* 0,1 *and* 2 *a pattern is found, each of which leads to a cube:*

0) $(0, 1, 1) \rightarrow \overline{i}_2[0] \cdot i_1[0] \cdot s_1[1]$
1) $(0, 1, 1) \rightarrow \overline{i}_2[0] \cdot i_1[0] \cdot s_1[1]$
2) $(0, 0, 0) \rightarrow \overline{i}_2[0] \cdot \overline{i}_1[0] \cdot \overline{s}_1[1]$

No other patterns are found at later time steps. The resulting property is the sum of the cubes, i.e.

$$p(T) = \overline{i}_2[0] \cdot i_1[0] \cdot s_1[1] + \overline{i}_2[0] \cdot \overline{i}_1[0] \cdot \overline{s}_1[1].$$

The number of time relations is large, since each of the signals can be assigned to any time step from 0 to $t_{max}-1$. This leads to $t_{max}^{|I|}$ time relations. But the search space can be pruned by using the following rules:

1. At least one time reference must be zero, otherwise the same time relation is considered more than once with a constant offset.
2. No signal is considered twice in the same time step. If a signal occurs more than once in I, different time steps are assigned to the different instantiations of the signal.
3. An input has no influence on a state bit or output, if it occurs at the last time step of the window.

Another observation helps to further reduce the search space. Given $|I|$ there can occur at most $2^{|I|}$ possible patterns with respect to a particular time relation. If all the possible patterns occur, the sum of the cubes returns the constant function 1 as a property, i.e. a property that is always valid. Thus, this time relation does not lead to a useful property and further scanning is skipped.

Currently the algorithm considers only one time relation for property generation. As a result no property that includes several time relations can be generated. This is the case, e.g. for existential quantification: in the propositional property this breaks down to a disjunction of several time relations.

The resulting property itself is represented by a *Binary Decision Diagram* (BDD) [3]. This introduces some abstraction from the cube representation, e.g. don't cares are easily determined. Because $|I|$ - the number of signals considered - is relatively small, BDDs are suitable to represent the property.

Using the following observation a "useful" property is chosen within the remaining time relations: If a time relation does not reflect the behavior of the signals, the occurring patterns seem randomly distributed - the number of different patterns is close to $2^{|I|}$. If a time relation exactly represents the behavior only a few patterns occur. Therefore those time relations with few patterns are considered "useful". Further pruning is based on this observation, only the time relations with the least number of patterns survive. By this, the evaluation order of time relations can lead to more or less efficient pruning. In this context increasing and decreasing order are distinguished. For the increasing order at first time step 0 is assigned to all signals at first, then incrementally all time relations are iterated until all signals get time step $|t_{max} - 1|$ assigned. Decreasing is the opposite order.

Furthermore the property generation can be guided by restricting signals to certain time steps or values.

The techniques introduced in this section provide means to generate properties from traces and to choose "useful" properties. Improvements of these techniques lead to an even higher quality of the analysis of the testbench. This is focus of current work.

IV. Experimental Results

The method is suitable for a tight integration of simulation based verification methods with formal proof techniques. Given a trace and a set of signals a number of properties is generated. Each of these properties is valid on the trace. If the whole testbench is used as the trace, but the resulting property for some signals is not valid for the design, a portion of the design was not yet tested.

In the following this is exemplarily shown for the small shift-register from Section II. Then, results for increasing length of the traces are shown in detail for one larger benchmark. Finally, sequential benchmarks from LGSynth93 are evaluated.

All experiments were carried out on an Athlon XP 2200+ system with 512MB RAM running Linux. Initially the properties were represented by cubes, these were then converted into BDDs. A simple BDD-based bounded model checker based upon CUDD [9] was used to check the validity of the resulting properties. The initial representation of properties by cubes is suitable because the number of patterns can not exceed $2^{|I|}$, where the number of signals $|I|$ is small. Only the property with the fewest patterns was retained as explained in Section III.

For the shift-register the signal tuple $I = (i_2, i_1, s_1, s_1)$ was used. The wanted property has to reflect the operating mode of "shift i_1 into s_1" and "keep the value of s_1", i.e. the property as a propositional formula should be:

$$p = \overline{i}_2[0] \cdot (i_1[0] \equiv s_1[1]) + i_2[0] \cdot (s_1[0] \equiv s_1[1])$$

Property generation was carried out for a trace of length 30 that was randomly generated. At first only 5 time steps, then 10, 20 and finally the whole trace were considered. For these cases the generated property became increasingly better as the trace covered more and more of the functionality. For all trace lengths the correct time relation was found, where i_1 and i_2 were picked at time step 0 and the two instantiations of s_1 at time 0 and 1.
$t_{cyc} = 5$: Only the non-shifting-mode ($i_2 = 0$) was covered, but only for the case $i_1 = 0$ and $s_1 = 0$:

$$p_5 = \overline{i}_2[0] \cdot \overline{i}_1[0] \cdot \overline{s}_1[0] \cdot \overline{s}_1[1]$$

$t_{cyc} = 10$: Both operation modes were covered, but only for certain values of i_1 and s_1:

$$\begin{aligned} p_{10} = \quad & \overline{i}_2[0] \quad (\overline{i}_1[0] \cdot \overline{s}_1[1] + i_1[0] \cdot s_1[1] \cdot \overline{s}_1[0]) \\ & + i_2[0] \quad \cdot \overline{i}_1[0] \cdot \overline{s}_1[0] \cdot \overline{s}_1[1] \end{aligned}$$

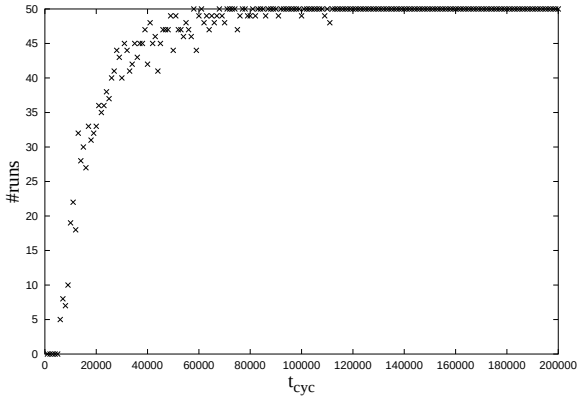


Fig. 4. Runs resulting in a valid property for 'misex3'

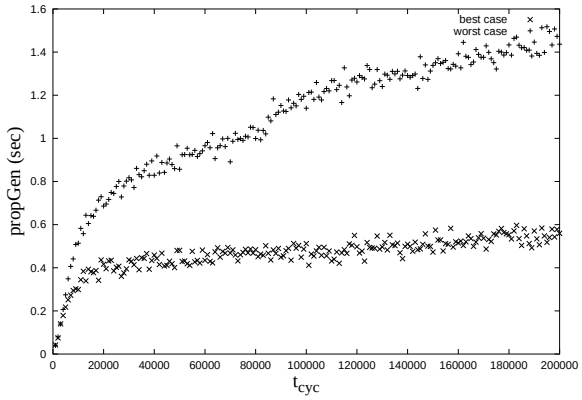


Fig. 5. Time needed for property generation for 'misex3'

$t_{cyc} = 20$: The shifting mode was covered completely, the non-shifting mode only for certain values of i_1 and s_1:

$$p_{20} = \quad \overline{i}_2[0] \quad \cdot(i_1[0] \equiv s_1[1])$$
$$+i_2[0] \quad \cdot(\overline{i}_1[0] \cdot \overline{s}_1[0] \cdot \overline{s}_1[1] + s_1[0] \cdot s_1[1])$$

$t_{cyc} = 30$: Both modes were covered completely and resulting was the property as given above.

This experiment shows how the amount of coverage achieved by the testbench is reflected by the property.

Figure 4 shows results for 'misex3'. This circuit has 14 inputs and 14 outputs. Circuit 'misex3' is combinational, but still property generation has to figure out the correct time relation. The diagram gives the number of runs resulting in a valid property for traces of different lengths. For each trace length 50 runs were carried out. As expected the number of runs resulting in a valid property increases with the trace length, because a better functional coverage is achieved. The time for property generation is very moderate as Figure 5 shows. The figure shows "worst case" (decreasing order of time relations) and "best case" (increasing order) as explained in Section III. Up to 40000 cycles often invalid properties were generated. In this case longer traces lead to better pruning of time relations (the ones that have more patterns than previous ones). For more than 40000 cycles onward mostly a valid property was generated. From this point additional cycles in the trace do not lead to more pruning, but to a linear increase of the time needed for scanning the trace.

Table I shows results for sequential benchmarks. For each circuit five runs were carried out. The parameter t_{max} was statically set to 4. For each run a random trace of 1 million clock cycles was generated and for I 7 signals were randomly chosen. The whole process of producing the random trace, generating the property and model checking was limited to 15 minutes. The time for property generation is shown for each run and also the result returned by the BDD-based model checker (v=valid, i=invalid, u=undecided, 1=all patterns occurred).

Even for the large number of 1 million clock cycles to be scanned for properties at most 420 seconds ('s838') are needed. Very often the runtime is even lower than 10 seconds. Especially runs resulting in valid properties are very fast, e.g. Runs 2, 4 and 5 for 's838' are much shorter than Runs 1 and 3, that result in trivial properties. This allows to use the tool on testbenches for large designs.

TABLE I
SEQUENTIAL BENCHMARKS, $t_{cyc} = 1000000$

Circ.	Run 1		Run 2		Run 3		Run 4		Run 5	
daio	v	1.03	v	0.66	v	0.95	v	4.10	v	0.96
gcd	u	164.39	u	303.57	u	331.01	1	22.45	u	167.84
mm4a	v	3.12	v	1.64	1	6.15	v	3.24	v	2.83
mm9a	u	86.25	1	1.30	u	72.92	1	26.35	1	7.31
mm9b	u	31.91	u	131.40	u	2.44	1	48.90	1	21.56
mult16a	1	0.32	1	0.08	v	1.43	1	0.13	1	0.09
mult16b	1	0.31	1	2.32	1	0.69	1	0.39	1	1.09
phase_d.	u	22.65	u	2.44	u	103.00	u	1.83	u	133.24
s1196	u	7.32	u	85.81	u	71.55	u	40.11	u	11.91
s1238	u	69.62	u	2.44	u	39.85	u	55.42	u	81.04
s1423	u	161.10	u	41.95	u	216.33	u	153.03	u	29.7
s344	1	8.31	v	2.39	1	1.53	1	2.82	v	4.26
s349	v	3.02	v	2.26	1	2.69	1	1.32	1	1.55
s382	1	0.78	1	1.94	1	0.59	1	0.49	1	1.28
s400	1	0.87	1	1.09	1	1.98	1	10.81	1	0.47
s420.1	1	1.87	1	34.88	v	44.71	v	24.96	1	7.23
s444	1	31.02	1	7.04	1	1.85	1	47.36	1	76.04
s526	1	1.62	1	1.60	1	3.18	1	1.07	1	0.93
s526n	1	6.24	1	135.65	1	1.52	1	3.55	1	53.39
s641	u	35.29	u	25.85	u	88.86	u	53.93	u	7.46
s713	u	78.50	u	1.51	1	2.80	u	0.88	u	118.19
s838.1	1	8.48	1	83.71	1	2.46	1	290.97	1	1.15
s838	1	416.28	v	1.32	1	291.80	v	1.25	v	1.21
s953	v	8.01	v	8.22	v	11.78	v	8.64	v	4.43
traffic	v	1.62	v	1.85	1	1.32	v	1.65	v	4.18

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*. Springer Verlag, 1999.

[2] J. Bormann and C. Spalinger. Formale Verifikation für Nicht-Formalisten (Formal verification for non-formalists). *Informationstechnik und Technische Informatik*, 43:22–28, 2001.

[3] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[4] Y. Hoskote, T. Kam, P. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Design Automation Conf.*, pages 300–305, 1999.

[5] S. Katz and O. Grumberg. Have I written enough properties - a method of comparison between specification and implementation. In *CHARME*, pages 280–297, 1999.

[6] J.P. Marques-Silva and K.A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Int'l Conf. on CAD*, pages 220–227, 1996.

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

[8] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engeneering an efficient SAT solver. In *Design Automation Conf.*, pages 530–535, 2001.

[9] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.1*. University of Colorado at Boulder, 2001.