Design Understanding by Automatic Property Generation

Rolf Drechsler Görschwin Fey

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{drechsle,fey}@informatik.uni-bremen.de

Abstract— Only a concise synthesis and verification flow allows to cope with complex circuits and systems consisting of several million components. In the meantime, verification has become the dominating factor causing up to 80% of the overall design costs. But still verification can only be applied if a formal model exists. Therefore the initial translation of the specification - given as a workbook in natural language - to a formal description on register-transfer level (RTL) is usually not checked. By this, current verification approaches do not provide any *design understanding*.
In this paper we propose a new approach that allows automatic generation of properties for a given design. These properties are formally verified using model checking. The resulting properties are translated into a description that is easy to read and to understand for the designer, who can add this description to the set of properties or a testbench. The methodology - independently of the designer and verification engineer - provides design understanding and by this significantly contributes to the quality of the process.

I. Introduction

In modern circuit and system design, verification becomes the major bottleneck and in the meantime dominates the cost for synthesis. Up to 80% of the overall design costs are due to verification. The classical approach to verification is based on simulation, but creation of large testbenches and also the (manual) checking or creation of monitors are very time consuming and error prone. But pure simulation is not sufficient to check the correct functional behavior, i.e. the coverage that is obtained is too low (see e.g. [1]).

In addition to simulation, formal verification techniques have been proposed and in the meantime are used in many industrial flows. In some cases formal methods have even replaced simulation, e.g. in equivalence checking [6]. As recent publications show, not only the quality of the verification improves, but also the costs in terms of man power can decrease [11]. Instead of setting up testbenches to check functional correctness, properties are mathematically proven to be correct and hold in any situation. Beside equivalence checking and property checking, approaches based on symbolic simulation or assertion checking proved to be very powerful in practice [5]. Unfortunately all these techniques can only be applied if a *formal description* of the circuit exists - either on behavioral level or on RTL.

But with increasing design complexity it becomes more and more important to get an understanding of the design, i.e. to check whether the implemented formal model corresponds to the intention and ideas of the person who wrote the initial specification (usually in form of a workbook). This specification is commonly given in natural language and by this may contain inconsistencies, non-precise descriptions or even contradicting requirements.

In this paper we present a new approach that is based on formal techniques. In contrast to previous techniques the goal is not to prove the correctness of given formulas or properties, but to automatically generate properties that are shown to the designer. The properties are generated based on a set of signals and a simulation trace. The set of signals is selected by the user, while the trace is derived from a testbench or random simulation. Generated properties are translated into a readable format such that they can easily be understood. The tool is based on a pattern matching technique and can be configured to generate valid, but also invalid properties. By this the designer gets feedback about the functional behavior of the system and can "discuss" with the tool. In contrast to previous approaches this method focuses on design understanding. It can be applied without having a formal model of the specification and by this targets at design verification instead of implementation verification - in contrast to most other tools. This difference will be explained in more detail below.

The paper is structured as follows: In Section II we describe in more detail the underling ideas and the methodology. First, the classical design method is briefly reviewed and the resulting verification approaches are discussed. Then, the new technique is presented and advantages and disadvantages are discussed. Details on the implementation of the tool are given in Section III. The pattern matching algorithm is discussed and criteria for selecting "useful" properties are given. In Section IV experimental results are given and finally the paper is summarized.

II. Methodology

In this section we first briefly review the classical design flow and resulting implications for verification techniques. Then, the new methodology is introduced. The integration into the design flow is described and resulting benefits are discussed.

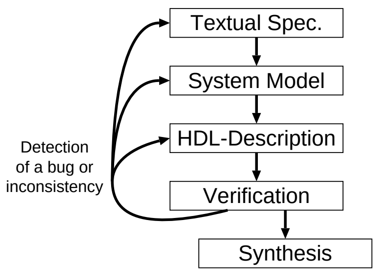


Fig. 1. Design process

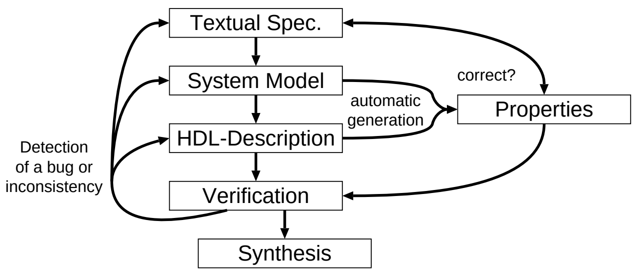


Fig. 2. Proposed methodology

A. *Classical Verification*

Still, verification is downstream in the design process as shown in Figure 1:

1. The initial idea is written down in a textual specification. Even though the specification might contain some formal parts, it is usually given in a natural language. This specification is then handed to the design team.

2. The textual specification is formalized and used to build a system model. This can be on a behavioral level or on RTL. Usually a common programming language like C or C++ is used to implement the system model.

3. The model is then coded in a hardware description language (HDL). This HDL model is built according to the textual specification.

4. The HDL model is checked

    - against the system model by means of testbenches or
    - against the specification by means of property checking.

5. The HDL model is synthesized.

Following the usual notation, the verification of the specification is addressed as *design verification*, while *implementation verification* covers the steps from the first formal description down to the final layout (including various stages of equivalence checking, etc.).

Design verification is only addressed in Step 4.

Remark 1 *Since the main focus of this paper is on the verification of the design entry, all verification issues related to design implementation, like e.g. verifying the correctness of the synthesis process, are not further considered in the following.*

Moreover in a large design project frequent changes of the specification may occur. These changes are incorporated into the design by repeating the steps shown above.

This process leads to a late detection of failures within the design process. When a failure is detected even modifying the specification can be necessary. This causes long delays during the design process.

B. *New Approach*

Here, we propose to incorporate techniques for formal verification at earlier stages of the process. As soon as first blocks can be simulated at a cycle accurate level properties should be automatically deduced. This helps to get insight and offers a different view at the design. By this, conceptual errors as well as coding errors can be detected earlier. Also the deduced properties can be used as a starting point for formal verification and by this reduce the time needed to set up the verification environment.

The idea is to provide a tool to the designer that allows to gather more insight into his own design. For an overview see Figure 2. Usually the system model already contains the cycle accurate I/O-behavior of most blocks. Therefore, as soon as first portions of the design can be simulated at the accuracy of clock cycles, formal properties are derived from the given description, i.e. from the system model or the HDL description. These properties exhibit some behavior of the design. The designer or a verification engineer has then to decide, if the property is correct or not. I.e. the compliance of the property with the textual specification has to be checked. If the property is found to be valid it can be used as a starting point for formal verification. If the property is incorrect, either the given simulation trace does not show all behaviors of the given block or the block is erroneous. By this, a direct feedback between textual specification, system model, HDL description, and verification is established.

This feedback between the different design stages helps to improve design quality. Instead of only assuming a property the designer can explicitly search for a property and check the compliance with the specification. Moreover pulling verification methods into the earlier stages of the design process enables an early detection of design errors. A mismatch between specification and HDL model is usually only detected during verification. But the proposed method unveils this mismatch already while coding a block. The iteration becomes superfluous.

Still the verification step can not be discarded. But as the set up of testbenches, also properties are already targeted during coding or even when only the system model is given. As soon as a trace is guaranteed to exhibit all important behavior the corresponding stimuli become part of the testbench. The same holds for deduced properties. These are added to the property suite for later formal verification. By this, a starting point for formal verification is created during coding already and time is saved, as not all properties have to be written manually.

C. Discussion

The presented methodology leads to a different aspect of the design than other techniques do. At a similarly early stage of the circuit design phase usually only lint checking or assertion checking are applied. But in case of linting only general properties that guarantee e.g. correct handling of arrays are checked. Using assertions semantic checks can be carried out by means of powerful properties [8]. But in this case the designer has to write the assertions himself, i.e. an assumption about the design is formulated as a corresponding property. The proposed method allows to retrieve insight from a simulation trace. No further knowledge about the design is needed.

In the software domain a similar methodology has been presented in [9]. In that case Java programs have been considered and execution traces have been searched for a set of predefined invariants. These invariants were afterwards statically checked. Here, the property detection works in a different way and is fitted to verification of hardware.

The properties considered in the following only argue about a fixed number of clock cycles, i.e. we make use of *Bounded Model Checking* (BMC) [2]. This allows to reduce the sequential problem of proving a property to a combinational one. Given a property that describes the behavior within l cycles 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 this combinational circuit.

III. Implementation

In this section the implementation of the proposed methodology is discussed in detail. For a better understanding the tool for property deduction is explained before the selection of properties is considered. Finally, the application of property deduction and the integration with methods for formal verification is studied.

A. Automatic Property Generation based on Pattern Matching

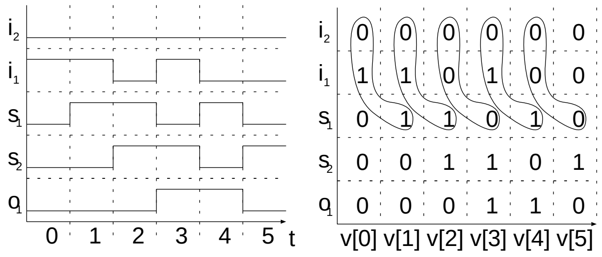
In the following the necessary basic notions to make the paper self-contained are given. These are circuits, traces and the type of properties considered.

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 $v[0], \dots, v[t_{cyc} - 1]$. 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 3(a). These can directly be mapped into the vector notion which is shown in Figure 3(b). Thus, necessary data can be generated from any simulation trace e.g. the widely used VCD format.



(a) Waveform

(b) Vector representation of the waveforms

Fig. 3. Representation of simulation traces

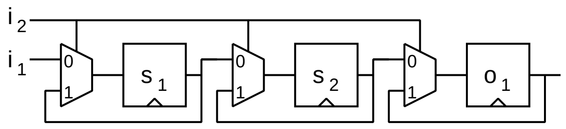


Fig. 4. 1-bit-shift-register

Note, that internal signals can not be considered using this notation, but the extension is straightforward. 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 4. 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 in three consecutive time steps, the value of i_1 in the first time step equals o_1 in the fourth time step" or written as a formula:

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

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. [3]). 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.

```
        PropGen($I, t_{max}, v[0], \ldots, v[t_{cyc}-1]$)
(0)     foreach time relation $T$
(1)       $p(T) = 0$
(2)       foreach time step $0 \leq t < t_{cyc}$
(3)         $pat$= getPattern($I$,$T$,$v$,$t$);
(4)         $p(T) := p(T) + pat$
(5)       end
(6)     end
```

Fig. 5. Sketch of the property generation

B. *The Basic Procedure*

The generation of properties uses 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.

The concept to deduce properties from traces is similar to the approach introduced in [7]. The basic procedure is given in Figure 5: Given are a tuple of signals I and a maximal window t_{max} for the properties to be generated as well as a simulation trace v of length t_{cyc}. In the property each of the signals is assigned to a particular time step in the window, that 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 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. Values for a pattern are determined by *getPattern*. Then, the behavior reflected by the pattern is included in the property.

Technically, the extracted pattern is a cube pat. This is joined with all previous cubes for the current time relation by calculating $p(T) := p(T) + pat$. The cube is given by transcribing the pattern into 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. Therefore one cube determines one valid assignment to the signals, the disjunction of all these cubes leads to the property $p(T)$.

Example 2 *Consider the trace given in Figures 3(a) and 3(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* $(i_2[t], i_1[t], s_1[t+1])$*. These patterns are indicated in Figure 3(b) by bubbles. At time steps* 0,1 *and* 2 *a new pattern is found, each of which leads to a cube:*

0) $(0, 1, 1) \rightarrow \bar{i}_2[0] \cdot i_1[0] \cdot s_1[1]$

1) $(0, 1, 1) \rightarrow \bar{i}_2[0] \cdot i_1[0] \cdot s_1[1]$

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

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

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

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 using the following rules:

1. At least one time reference must be zero.

   Otherwise the same time relation is considered more than once. I.e. a time relation starting at 0 would be shifted to a starting point greater than 0.

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.

   Otherwise one value of the pattern would be considered twice, which is superfluous.

3. An input is never considered in the last time step of the window.

   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 possible patterns occur, the sum of the cubes returns the constant function 1 as a property, i.e. a property that is trivially true. Thus, this time relation does not lead to a useful property and further scanning the trace is skipped.

A restriction of the current algorithm is that only one time relation is considered for generation of the property. 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 not represented by explicit cubes, but symbolically by a *Binary Decision Diagram* (BDD) [4]. This can not lead to memory blow-up because $|I|$ - the number of signals considered - is relatively small. Additionally BDDs introduce some abstraction from the cube representation, e.g. don't cares are easily determined.

C. Selection of Properties

C.1 Choosing a Useful Property

For each time relation that is not pruned by the rules shown above, a valid property is generated. Then it has to be decided which of this large number of properties are "useful". This obviously can not be done fully automatically. But indeed some help can be provided.

As stated at the end of the last section a property that is trivially true, i.e. equal to constant 1, is of no use. Also if the relation between some signals in time is determined by the underlying circuit, the number of occurring patterns is small compared to $2^{|I|}$. In the other case, if the relation of the signals is not determined by the circuit, the values in the patterns seem to be randomly distributed and thus the number of occurring patterns is close to $2^{|I|}$.

Example 3 *Consider the shift-register given in Figure 4, $I = (i_2, i_1, s_1)$ and a trace reflecting any state and any input sequence for the shift-register. For the time relation given in Example 2 the following holds true:*

- *"If $i_2[t] = 0$, then $s_1[t+1]$ is equal to $i_1[t]$". This breaks down to two cubes representing $\overline{i}_2[0] \cdot (i_1[0] \equiv s_1[1])$.*
- *"If $i_2[t] = 1$, then $s_1[t+1]$ and $i_1[t]$ are independent", leading to the cube $i_2[0]$.*

This are six cubes in total.

Now, consider a time relation, where the value of s_1 is taken at a time step greater than 1. In this case the value can not be predicted from the other two values. Therefore all patterns occur and the property becomes the constant function 1, i.e. trivially true.

This observation can be used to order the properties generated from the trace. Resulting properties are ordered by decreasing numbers of different patterns that were observed. This ranking is used to decide about the "usefulness" of properties. The ranking also helps to prune evaluations of other time relations. Only a limited number of properties with the least number of patterns is retained.

C.2 Guided Property Generation

When being confronted with a large design more focused properties can be useful. This can be formulated as an assumption to restrict the property generation. In case of the shift-register there are two different modes of operation. Either $i_2 = 0$, i.e. the register shifts at each clock cycle or $i_2 = 1$, i.e. the register keeps the current state. The method so far only allows to generate properties for any relation between the signals. Often a property focused to a certain mode of operation can be more desirable.

This focusing can be done by an extension of the property with an assumption. This assumption restricts some signals in I to a certain value, to the value of another signal in I or to a particular time step within the window. Only a pattern that

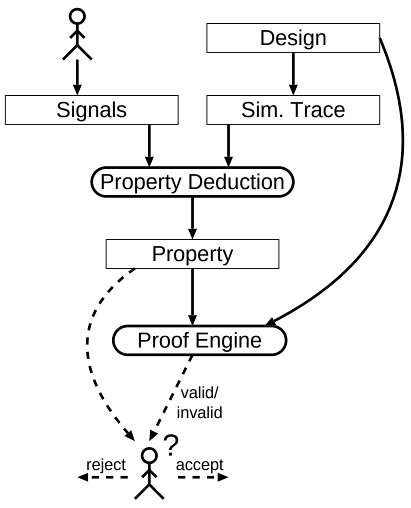


Fig. 6. Application of property deduction

does not violate the assumption is included in the property. The assumption can also be rewritten as a propositional formula a. Thus, the resulting property becomes $P(T) = a \rightarrow p(T)$, where $p(T)$ is generated as above, but only from patterns fulfilling the assumption.

Example 4 *Assume, that in case of the shift-register, the operating mode for shifting is particularly interesting. Therefore the assumption $a = (i_2[t] \equiv 0)$ is used. In this case only cubes where $i_1[t] \equiv s_1[t+1]$ are collected. As a result the property $P = (i_2[0] \equiv 0) \rightarrow (i_1[0] \equiv s_1[1])$ is generated.*

Currently simple assumptions are allowed, e.g. the restriction of a signal to a certain value or to the value of another signal. Also a signal can be restricted to be considered only at a particular time step within the window of the property. More complex constructs can easily be allowed by extending the input language used for assumptions. On a given trace the check, if an assumption holds, always breaks down to fast pattern matching.

C.3 Property Completion

In cases where a large number of signals or states is considered, the given simulation trace can not cover the complete behavior of the design. In this case the property that is extracted from the trace is not valid within the design. But formal techniques can be applied to complete this invalid property and retrieve a valid one.

For this purpose the engine that is used to prove properties on the design must have the capability to find all counter-examples, if the property is invalid. This is true for example in case of BDDs or if a complete SAT solver is used. Each counter-example is a pattern that was not found in the trace. The counter-example is included into the property in order to become valid. When the added counter-examples are shown to the designer, this provides a feedback about behavior that was not covered by the simulation trace. The aim of understanding the design benefits from this feedback.

TABLE I

SEQUENTIAL BENCHMARKS, $t_{cyc} = 100,000$

Circ.	Run 1					Run 2					Run 3				
	#rel	time	res.	#pat	#diff	#rel	time	res.	#pat	#diff	#rel	time	res.	#pat	#diff
daio	1	0.42	v	18	-	1	0.21	v	24	-	3	0.36	v	22	-
gcd	1	8.11	u	47	-	1	14.08	u	64	-	12	31.89	u	71	-
mm4a	2	1.93	i	91	a 9	1	1.32	v	56	-	2	0.53	v	50	-
mm9a	1	4.86	u	106	-	3	14.49	u	117	-	1	1.11	u	78	-
mm9b	2	11.26	u	105	-	18	1.95	u	56	-	6	1.31	u	56	-
mult16a	0	0.08	1		-	0	0.27	1		-	0	0.5	1		-
mult16b	81	5.96	v	96	-	0	2.25	1		-	0	2.03	1		-
phase_d.	2	5.81	u	10	-	109	8.05	u	6	-	115	8.97	u	6	-
s1196	1	62.07	u	70	-	5	50.42	u	57	-	1	1.79	v	49	-
s1238	3	2.53	u	54	-	2	1.96	u	66	-	1	4.05	i	94	a 18
s1423	4	27.36	u	79	-	1	10.93	u	106	-	10	19.42	u	34	-
s344	18	5.89	i	127	a 1	186	10.64	v	48	-	17	1.45	i	60	a 60
s349	0	0.12	1		-	1	4.22	i	108	a 20	27	5.38	i	65	a 15
s382	72	19.15	i	17	a 63	40	16.55	i	7	a 49	48	3.37	i	7	a 24
s400	17	0.9	i	6	a 54	18	1.45	i	18	a 48	18	1.53	i	5	a 32
s420.1	0	0.23	1		-	0	0.26	1		-	0	4.53	1		-
s444	26	11.24	i	16	a 112	10	1.43	i	4	a 40	234	35.6	i	5	a 25
s526	16	117.81	i	5	a 91	2	35.32	i	22	a 106	38	4.23	i	15	a 51
s526n	9	1.78	i	5	a 91	71	6.88	i	7	a 121	22	1.95	i	14	a 58
s641	653	49.5	u	32	-	0	1.78	1		-	2	24.65	u	87	-
s713	453	35.76	u	32	-	5	7.96	u	94	-	3	25.38	u	78	-
s838.1	551	41.34	i	32	a 96	96	7.58	i	16	a 112	308	38.97	i	64	a 64
s838	695	46.95	v	16	-	116	7.59	v	4	-	83	7.88	v	4	-
s953	74	15.82	v	33	-	5	2.34	v	14	-	7	4.16	v	36	-
traffic	3	0.96	v	7	-	17	2.73	v	40	-	2	0.66	i	29	a 7

D. *Application of Property Deduction*

The work flow to apply property deduction is shown in Figure 6. As a starting point the design and simulation traces are available. Then a tuple of signals is selected that have to be related to each other. This is handed to the automatic property deduction and a property is retrieved that is valid on the trace. The property is passed to a proof engine that returns validity or invalidity of the property. All information is provided to the designer who decides, if the property is to be accepted or rejected. In case of acceptance the property is added to the property suite and if necessary corrections to the design are made. If the property is rejected another deduced property with lower ranking can be considered or a different tuple of signals can be chosen.

During the decision the techniques introduced previously aid the designer. The simplification of properties leads to the more general behavior of the design and by this the decision is easier. Opposed to that the completion of a property exhibits behavior that is not covered by the simulation traces and may reveal corner cases.

Remark 2 *In some cases it can be more instructive to review the property before the result of the proof engine is known. This allows to consider the property itself without being influenced by the verification result. Opposed to looking at the simulation trace, considering the property allows to focus on the functional relation of the signals more easily.*

Altogether this establishes an interactive process that offers different views at the design. Such a new perspective is an opportunity to increase the insight and to understand the design.

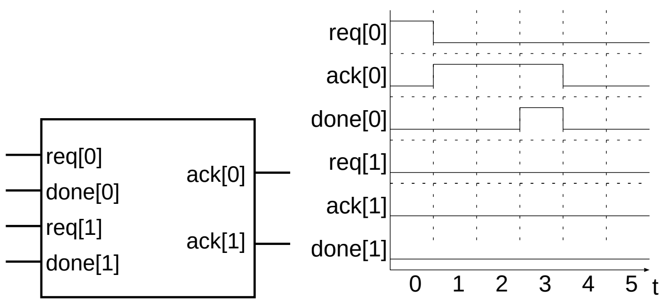
IV. EXPERIMENTAL RESULTS

Two types of experiments are carried out in the following. At first the efficiency of property generation is evaluated by using the LGSynth93 benchmark set. Then, a case study shows the benefits of property generation while implementing an arbiter.

A. *Benchmarks*

The focus of this work is to give some information about the relation between deduced properties and the design. For this purpose a number of sequential circuits from the LGSynth93 benchmark set was considered. For each of these circuits the results of three runs are shown in Table I. In all cases traces of 100,000 clock cycles were considered. The tuple of seven signals was chosen randomly and t_{max} was set to 4. A simple BDD based proof engine was implemented and tightly integrated with the property generation to decide the validity of a property.

For each run several data is reported. Column “#rel” gives the number of time relations with a minimum number of patterns, i.e. the number of properties with the highest ranking according to Section III.C.1. The time in CPU seconds needed to



(a) Block diagram (b) Trace

Fig. 7. The arbiter

collect the data and by this deduce properties is shown in column “time” (AMD Athlon XP 2200+, 512 MB). The following three columns give information about the first property of those with highest ranking. Column “res” states, if the property was valid (v), invalid (i), trivial (1), or the proof engine exceeded the time limit or memory limit and the property was left undecided (u). In column “#pat” the number of patterns included in this property is shown. Finally, Column “#diff” gives the number of added patterns in order to turn an invalid property into a valid one. This case is marked by the preceding letter “a”. A trivial property was not further considered.

As can be seen from the table, the number of time relations with a high ranking is rather small in most cases. This strongly underlines the feasibility of the interactive application of the tool by going through the highest ranked properties. The large number of properties that were left undecided is due to the proof engine that was based on BDDs only. Instructive is the number of patterns that were added to valid properties. This shows, that in some cases the simulation only covered a small fraction of the total behavior of the signals. When a large number of properties with high ranking occurs, the number of added patterns is an additional hint for selecting useful properties.

Note, that the experiments in this section are a worst case scenario for property deduction:

- The tuple of signals is choosen randomly.
- The window length is fixed.
- The simulation trace was generated by random stimuli.

In the usual application scenario a designer would choose a set of appropriate signals and a window length. Often stimuli provided by a testbench could be used for property deduction. A more realistic scenario is shown in the following case study.

B. Case Study: Arbiter

The benchmark results above show the efficiency of property generation. But these examples do not show the quality of the generated properties or the feasibility of the approach. As a case study a simple arbiter was coded and checked by means of property generation.

```
1  module theArbiter (clock,ack,done,req);
2    parameter    IDLE=0, BUSY=1;
3    input        clock, reset;
4    output [1:0] ack;
5    input  [1:0] req, done;
6
7    reg    [1:0] ack;
8    reg          state;
9
10   wire   [1:0] resolve, acquire;
11
12   assign resolve[0]= req[0];
13   assign resolve[1]= !req[0] & req[1];
14 //assign acquire= (ack[0] & req[0])
                    | (ack[1] & req[1]);
15 //assign acquire= (ack[0] & done[0])
                   | (ack[1] & done[1]);
16   assign acquire[0]= ack[0] & !done[0];
17   assign acquire[1]= ack[1] & !done[1];
18
19   always @(posedge clock)
20     case (state)
21       IDLE: if (req!=0)
22               begin
23                 ack= resolve;
24                 state = BUSY;
25               end
26       BUSY: if (done!=0)
27               begin
28                 ack= acquire;
29                 state= IDLE;
30               end
31     endcase
32 endmodule
```

Fig. 8. Code of the arbiter

The arbiter manages the access of two masters to a bus. Conflicts are resolved by a priority scheduling. There exists a request input (`req`) and a done input (`done`) as well as an acknowledge output (`ack`) for each master. Figure 7(a) shows a block diagram of the arbiter with two masters. An example for a request from master 0 is shown in Figure 7(b). By setting `req` a master signals the need to access the bus. Then, the arbiter sets `ack`, if the bus is not in use and no request of higher priority occurs and `ack` is kept. Finally, the master sets `done` to release the bus again.

The arbiter was coded using Verilog. VIS [10] has been used to generate a blif-file from the Verilog description. Then automatic property generation was used in the manner explained in Section D. Due to this process two errors were detected. The Verilog code of the arbiter is shown in Figure 8. Originally instead of lines 16 and 17 only line 14 was in place. In a first attempt to fix bugs this was replaced by line 15 and finally by lines 16 and 17. The detection of errors and reasons for the replacement are described in the following. For all calls of property deduction t_{max} was set to 2. The tuple of signals I considered is shown at the beginning of each paragraph.

$I = ($`req[0], ack[0]`$)$:
At first the relation between `req` and `ack` for the master with highest priority was of interest. The set of signals handed to property generation consisted only of `req[0]` and `ack[0]`. The first assumption was, that there is a dependency between this pair of signals. But indeed any pattern can occur. A trivial property was the result. Therefore the state was included in the set of signals.

$I = ($`req[0], state, ack[0]`$)$:
The first version of the arbiter contained line 14 instead of lines 16 and 17. This lead to an error: `ack` could be influenced by the behavior of `req` while the bus was `BUSY`. This error was resolved by replacing line 14 with line 15.

Now, additionally the influence of `done` on the other signals was of interest. Also a value of `ack` at another time step was taken into account.

$I = ($`req[0],done[0],state,ack[0],ack[0]`$)$:
The resulting property showed that the `ack` for the master was reset, even if the master did not release the bus by setting `done`. A possible solution is the replacement of line 15 by lines 16 and 17.

In the resulting property the `state` was taken at time step 1 instead of 0 as originally wanted. Therefore this signal was restricted to time step 0 and as a result the relation for the arbiter with highest priority was returned.

The case study showed a scenario for the application of property deduction and how errors can be revealed using this method. In the first query only a small tuple of signals was considered. This tuple was then successively enlarged to understand more relations. Checking deduced properties on the design and reviewing these gave a feedback, that led to the detection of design errors. During this process no direct interaction with the formal verification engine was necessary.

V. Conclusions

A methodology to improve design understanding by automatic property deduction has been introduced. The deduction of properties uses fast pattern matching on simulation traces and is therefore highly efficient. Resulting properties can be formally verfied, while the designer is relieved from writing properties. At the same time accepted properties serve as a starting point for formal verification. Even more important is the different view at the design provided by the generated properties. This can unveil behavior which remains hidden otherwise. By this the understanding of the design is improved. In turn the efficiency of the design process increases due to early detection of bugs or inconsistencies. In summary the advantages are twofold. Time for manually writing properties is saved and, even more important, a new perspective on the design is provided.

Future work will focus on the selection of properties and the front-end to present deduced properties to the user.

References

[1] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Design Automation Conf.*, pages 244–248, 2001.

[2] 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.

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

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

[5] R. Drechsler. *Advanced Formal Verification*. Kluwer Academic Publishers, 2004.

[6] R. Drechsler and S. Höreth. Gatecomp: Equivalence checking of digital circuits in an industrial environment. In *Int'l Workshop on Boolean Problems*, pages 195–200, 2002.

[7] G. Fey and R. Drechsler. Improving simulation-based verification by means of formal methods. In *ASP Design Automation Conf.*, pages 640–643, 2004.

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

[9] J.W. Nimmer and M.D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[10] The VIS Group. VIS: A system for verification and synthesis. In *Computer Aided Verification*, volume 1102 of *LNCS*, pages 428–432. Springer Verlag, 1996.

[11] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. A 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.