Property Mining using Dynamic Dependency Graphs *

Jan Malburg[‡], Tino Flenker[†], and Görschwin Fey[‡ †]
[‡]German Aerospace Center, Institute of Space Systems, 28359 Bremen, Germany
[†]University of Bremen, Institute of Computer Science, 28359 Bremen, Germany
Email: Jan.Malburg@dlr.de, {flenker, fey}@informatik.uni-bremen.de

Abstract — We present a technique to automatically generate SystemVerilog-Assertions from designs using dynamic dependency graphs. We extract relations between signals of the design using only a few simulation runs, which drastically reduces the required number of use cases compared to other approaches. Additionally, unlike previous approaches, we do not use expression templates to establish those relations. We abstract from the concrete use cases by inserting symbolic values and by merging similar conditions in time. A model-checker verifies the correctness of the generated properties. The evaluation shows that our approach is able to create more expressive properties than state of the art techniques, while requiring less simulation data.

I. Introduction

In modern chip design, especially for *System-on-Chip* (SoC)-designs, the final chips are created by combining different functional blocks. Those blocks can be newly created blocks, blocks from previous designs, or blocks licensed from other companies [1]. Due to increase of design complexity and integration of more and more functionality into a single chip, the amount of reused and licensed blocks is steadily increasing. In order to correctly use those blocks a good knowledge of their behavior is required. Such information about the behavior of functional blocks includes definitions of correct inputs to the block, latency, changes of the internal state, and the expected outputs. In the cases where the original designer of the design is not available, the user must get the information from the documentation of the design. There are different ways to describe the behavior of functional blocks, e.g., informal descriptions in natural language or in a formal and standardized format. One example of such a standardized format are SystemVerilog-Assertions. Formal and standardized specifications have several advantages; they are unambiguous, they can be automatically processed by different tools, and they can be automatically checked for correctness. Further, they can be used to create test cases for the design [2].

In this work, we present an approach to automatically generate formal properties for a given design. As discussed in [3] generated properties are useful for understanding a design, regression testing, or detecting holes in testbenches. In the first step, our approach simulates a given set of use cases. In contrast to other approaches, even a small number of use cases suffice. The use cases may be provided by the user, or automatically created. From these simulation runs *Dynamic Dependency Graph*s (DDG) are created. Utilizing the DDGs an initial set of potential properties for the design is created. For creating this initial set of properties, symbolic representations of the observed concrete values are used. The use of symbolic values already provides an abstraction from the concrete values. By combining properties with similar sequences we can abstract temporal behavior over unbounded time ranges. In the end a model checker decides whether the abstracted properties hold for the design.

Our contribution is the first approach to generate properties from DDGs for hardware. Starting from use cases ensures guidance to the property inference. Using DDGs allows for powerful generalization. In contrast to previous approaches, which required a large amount of use cases and templates to establish relations between signals of the design, we only need a very small number of DDGs and no templates to find those relations. This makes our approach complementary to previous work.

Experiments with our approach showed that we can create very good properties, even from a small set of simulation data. In contrast to approaches using data mining, our technique further allows to generate word-level properties utilizing the whole set of SystemVerilog operators.

This paper is structured as follows. In Section II related work is shown. Preliminaries are defined in Section III. Next, our approach is presented in Section IV. Section V evaluates our approach and Section VI concludes the paper.

II. Related Work

Dynamic invariant mining is a technique to compute likely invariants from example executions of a system. A well known tool in this area is Daikon [4]. Daikon computes invariants by instrumenting the design under consideration. Next, the design is executed using a set of use cases. Afterwards, statistical analysis is performed on the gained data to obtain the invariants. This analysis searches for values, which satisfies a set of property templates. Our approach differs from the Daikon approach in three major facts. First, we use a model-checker to verify the invariants to ensure the correctness of the generated properties. Second, our approach uses the concrete computa-

*This work was supported in part by the European Union (IMMORTAL project, grant no. 644905), by the University of Bremen's Graduate School SyDe, funded by the German Excellence Initiative and by the German Research Foundation (DFG, grant no. FE 797/6-1).

tion of the values given by the DDG to compute the invariants and not only the observed values. Finally, our approach does not need expression templates, as the expressions are extracted from the DDGs.

In [5] Nimmer and Ernst combine Daikon with the static program checker ESC. In this combination the ESC is used to model-check the invariants generated with Daikon. This combination lifts the limitation of Daikon that the invariants are not verified, but still the computation is only based on comparisons of recorded values without any guarantee that they are related in any way.

Gabel and Su describe the property mining tool Javert [6]. Javert computes temporal properties describing the correct usage of software artifacts. More precisely they compute a regular expression, which defines the correct sequence of function calls. However, their approach cannot simply be applied to hardware designs. They use function calls as target for their tracking and specification. For hardware designs at the *Register-Transfer-Level* (RTL) the inputs to a design are not provided as function calls. In fact the interface of a hardware block does not show which inputs decide the functionality and which are data for the function, data- and control-inputs may even be time multiplexed, or inputs may be ignored at given times. Applying the Javert approach to RTL-design would require a large manual effort to describe the requests for functionality and when they are valid. This contradicts the goal of an easy to use automated approach.

Vasudevan et. al. present the GoldMine tool in [7]. The tool automatically generates properties for a given RTL-design. The tool uses simulation traces, formal verification, and static code analysis. GoldMine applies use cases to a design and stores signal values during the simulation. In the next step a decision tree based learning algorithm computes candidate relations between signals. The resulting properties argue about equality and non-equality of signal with constants. GoldMine argues only over single bits. Afterwards a designer may mark whether she considers the generated property useful which supports the learning algorithm to direct the generation process. The generated properties are mostly used for regression testing; however they can also be checked against the specification to uncover incorrect behavior. In contrast to GoldMine which uses simulation data for property generation, we use the DDG to take control- and data-flow of the design into account. Further, our approach can create word-level properties and uses a more expressive set of operands.

The approach in [8] relies on templates and statistical analysis to infer formal properties from simulation traces. Similar to our approach, formal verification ensures the validity of properties. However, no reasoning from expressions as given by the DDG is considered.

The Inferno tool [9] analyzes a set of simulation traces to infer transactions of the design. For this, Inferno considers a user defined set of signals and the observed transitions of the values of these signals. First, this reduces a set of possibly large simulation traces into a small *Finite State Machine* (FSM). The Inferno tool searches for transactions in the simulation traces. Transactions are sequences of signal valuations which appear several times in the simulation trace and start and end in some boundary valuations. In contrast to our approach Inferno requires large amounts of simulation runs and requires additional information from the user, if the operations of the system are overlapping. Further, Inferno only describes the observed values and does not abstract symbolic values.

Dynamic symbolic execution is a technique to create use cases [10]. The technique first computes the symbolic constraint for an existing use case. This constraint describes the conditions under which a use case executes the same computation path. Then, one of those conditions is negated and from the constraint all subsequent conditions are pruned. In the next step a constraint-solver computes a new use case, fulfilling this new constraint. This new use case is expected to follow a different execution path. Dynamic symbolic execution and our approach have in common that they compute the path-constraints from concrete executions of the system, but how those path-constraints are used largely differs between both approaches.

III. Preliminaries

Let D be some design, given in a *hardware description language* (HDL), e.g., Verilog or VHDL. We say a *use case* for D is a sequence of assignments to D's primary inputs. The simulation of a use case is called a *run*.

Given a design D and a run r a DDG $G = (V, E)$ can be computed. A DDG consists of a set V of vertices, representing expressions, statements, and values of signals including primary inputs. In a DDG a statement, an expression, or a signal may have more than one vertex assigned to it representing different time points or instances. The edges of the graph E describe the dependency between those expressions, statements, and variables. Based on the definitions for the different types of dynamic slicing by Zhang, He, Gupta, and Gupta [11] we define three types of dependencies. The first type is *data-dependency*. A vertex v_1 is data-dependent on a vertex v_2, if v_2 is an operand of v_1 or if v_1 is a signal value and v_2 is the corresponding assignment to that signal. The second type is *full-dependency*. Full-dependency contains all data-dependencies and additionally also considers control-dependencies. A vertex v_1 is control-dependent on a vertex v_2 if v_2 is a control statement and v_1 is only executed because of control-decisions made by v_2. The last type is relevant-dependency. Relevant-dependency subsumes full-dependency and extends the definition of control-dependency such that a vertex v_1 representing a signal is also control-dependent on v_2 if v_2 is a control-statement and v_1 would have another value, if v_2 would have been evaluated differently.

IV. Inferring Properties

In our approach we compute properties for hardware designs using DDGs. The underlying idea is the following: Given a vertex v in the DDG the subgraph which only contains the elements on which v recursively dependent (backward slice), yields a correct property for v. The user defines a set of inputs and outputs for the properties, those may include internal

signals or registers of the design. The properties generated by the presented technique, describe a relation between the inputs and the outputs. By default our technique assumes the primary inputs of the design as inputs and the primary outputs of the design as outputs. If other registers/signals, for example the general purpose register of a CPU, shall be used, the user has to provide corresponding information to the approach. We create SystemVerilog [12] properties. The basic format is:

$$e_1\ r_1\ \#\#t_1\ e_2\ r_2\ ...\#\#t_{n-1}\ e_{n-1}\ r_{n-1}\ |->\ \#\#t_n\ o == e_n$$

Where o is an output, $e_1, ..., e_n$ SystemVerilog expressions, $t_1, ..., t_n$ non-negative integers and r_i is a formula how often e_i is repeated (described in Section IV.H). The basic steps of the approach are:

1. Creating DDGs
2. Computing initial properties
3. Combining similar properties
4. Splitting properties
5. Abstracting repetitions
6. Checking the properties

A. Notation

We use c as symbol for a constant. We denote a symbolic expression with the symbol e. A symbolic expression is an expression over constants and variables using a set of operators and the `$past`-function. In our case we allow all Verilog operators with the exception of the ternary operator, which we substitute with a conditional statement and an assignment to the correct branch. Early results of our approach showed that using the ternary operator in expressions leads to very poor and hard to read properties. We assume that a set of basic simplification rules are applied to the symbolic expressions, e.g., $a \vee 0 \Rightarrow a$. Further, let e_i be a symbolic expression using only inputs as variables and let e_{io} be a symbolic expression using inputs and outputs as variables.

We measure time in clock cycles. Given a variable v, v^T denotes the variable v at the clock cycle T. Given a symbolic expression or property x, and let $\mathbb{V} = v_1^{T_1}, ..., v_n^{T_n}$ be all occurrences of variables in x. Let $\mathbb{T} = T_1, ..., T_n$ be the corresponding set of all clock superscripts. For convenience we extend the superscript annotation to properties and symbolic expressions, thus we write x^T with $T = \max(\mathbb{T} \cup \{0\})$. Further, we define a transformation $x^T[c], c \in \mathbb{Z}$, such that each variable $v_x^{T_x} \in \mathbb{V}$ is replaced with $v_x^{T_x+c}$. We use $x^T[\downarrow]$ as shorthand for $x^T[-\min(\mathbb{T} \cup \{T\})]$, i.e., the transformation that causes the smallest value in $\mathbb{T}$ to become 0. Given a collection of symbolic expressions $E = \{e_1, ..e_n\}$, we define $E[c] = \{e_1[c], ..., e_n[c]\}$.

We say a *path-condition* is a symbolic expression of the form $e_i == c$ where e_i is the (symbolic) condition of a conditional statement and c is the concrete value to which this condition has evaluated. The following sections describe the steps of our approach in detail.

```
1  module ConditionalFlipFlop(
2    input  wire clk,
3    input  wire enable,
4    input  wire dataIn,
5    output reg  dataOut)
6
7    always @(posedge clk)
8      if(enable)
9        dataOut <= dataIn;
10 endmodule
```

Fig. 1. The code for our minimal example.

TABLE I
THE USE CASE FOR OUR EXAMPLE

clock tick	0	1	2	3	4
`enable`	1	0	0	1	1
`dataIn`	1	1	0	0	0

B. Creating Dynamic Dependency Graphs

First, we generate the DDGs. A code transformation is used to compute relevant-dependency. The basic idea of the code transformation is that whenever an evaluation of a branch could assign a variable ensure that all branches assign the variable, possibly in form of a self-assignment. After the code transformation is applied the design is instrumented, such that a DDG is generated while simulating the instrumented design.

Example 1. We use the conditional flipflop design given in Figure 1. The use case we will consider for our example is given in Table I.

Figure 2 shows the DDG for our example use case and design.

C. Computing initial properties

First, we annotate internal vertices if they are equal to an output vertex. For this we follow the assignments of the outputs backwards, while gathering all path-conditions under which those assignments were executed. We stop if we hit a vertex type other than assignments. Each visited vertex is annotated with the output as well as the path-conditions gathered until that point.

Example 2. Vertices v_{13} and v_{17} in Figure 2 will be annotated with $\texttt{dataOut}^3$ under the path-condition "$\texttt{enable}^2 == 0$" created by if_3. Vertices v_{12} and v_{16} will get two different annotations: First $\texttt{dataOut}^2$ under the condition "$\texttt{enable}^1 == 0$" created by $\texttt{if}_2$, second $\texttt{dataOut}^3$ under the set of conditions "$\texttt{enable}^2 == 0$" created by $\texttt{if}_3$ and "$\texttt{enable}^1 == 0$" created by $\texttt{if}_2$.

If we compute symbolic expressions of the type e_i^T we ignore those annotations. In the case of the $e_{io}^{T_1}$ type we chose the annotation with variable v^{T_2}, with the highest value T_2 such that $T_2 < T_1$. Now for each output signal o^{T_p} we compute an initial property as follows: We generate an expression e of the type e_{io} for the signal, while doing so we add each path-condition we encounter to a set C. "Encounter a path-condition" means that we either visit its corresponding conditional statement vertex or it is a part of an annotation we are using. Note, while we generate the expression for a path-condition we may encounter further path-conditions.

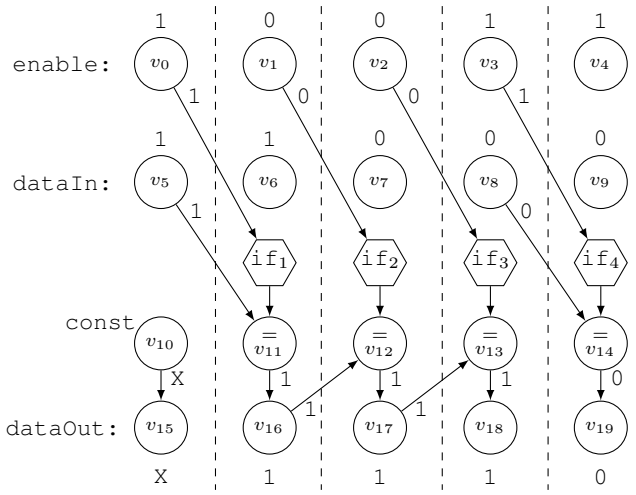

Fig. 2. DDG for the use case of Table I

We split C into a list of sets of path-conditions ordered by time $C^* = (C^{T_1}, ..., C^{T_n}), T_1, ..., T_n \in \mathbb{N} \cup \{0\}$ such that:

$$(\forall i \in \{1, .., n\} \forall c^{T_i'} \in C^{T_i}, T_i' = T_i) \wedge$$
$$(\forall i \in \{1, .., n-1\} T_i < T_{i+1})$$

The initial properties are of the form:

$$\left(\bigwedge_{c_1 \in C^{T_1}} c_1 \right) \#\#(T_2 - T_1) \left(\bigwedge_{c_2 \in C^{T_2}} c_2 \right), ...,$$
$$\#\#(T_n - T_{n-1}) \left(\bigwedge_{c_n \in C^{T_n}} c_n \right) |- > \#\#(T_p - T_n) o == e$$

In cases where $C^{T_i}[n] \equiv C^{T_{i+1}}[n-1] \equiv ... \equiv C^{T_{i+n}}$, i.e., for n clock cycles the conditions are identical modulo a moving time frame, we are using the SystemVerilog [*n] operator to shorten the property. We say n is the *repetition count* of the corresponding sets of path-conditions. We call $\{C^{T_i}, ..., C^{T_{i+n}}\}$ a repetition set. Further, we use the function `$past` to access variable values at previous clock cycles: Given a symbolic expression e^{T_k} and let v^{T_l} be a variable occurring in e^{T_k}, further let $T_l < T_k$ then we write that occurrence as `$past(v,`$T_k - T_l$`)`.

Our example would result in following properties:

```
enable |->##1 dataOut == $past(dataIn,1)
!enable |->##1 dataOut == $past(dataOut,1)
```

D. Combining similar properties

Next, we try to combine properties with respect to their repetitions. We decide whether we can combine two properties p_1, p_2 as follows: Let p_1', p_2' be copies of the original properties. Then for each repetition set $\{C^{T_i}, ..., C^{T_{i+n}}\}$ in one of the properties p_1', p_2' we remove $C^{T_i+1}, ..., C^{T_{i+n}}$ from that property and apply $\{C^{T_1}, ..., C^{T_i}\}[n]$. If then $p_1'[\downarrow] \equiv p_2'[\downarrow]$, p_1, p_2 can be combined. The assumption is that, if several path-constraints have such similarities, those will also hold for different numbers of repetitions.

To also include periodic behavior of the design, we represent the number of repetitions as a quadruplet $f = (\mathit{base}, \mathit{width}, \mathit{minSteps}, \mathit{maxSteps}) \in \mathbb{Z}^4$ in a (canonical) form such that each repetition count can be computed as: $\mathit{base} + n * \mathit{width}$ with $n \in \mathbb{Z}$, $0 \leq \mathit{minSteps} \leq n \leq \mathit{maxSteps}$, and $0 \leq \mathit{base} < \mathit{width}$. The reason behind this more complex representation of the number of repetitions, in contrast to a simple range of minimal and maximal amount of repetitions is that often for a hardware design there are functions which take a fixed amount of clock cycles to execute or read their inputs. Let $R_o = (r_1, r_2, ..., r_m)$ be the list of observed repetitions in ascending order without duplicates and let $\Delta = (\delta_1, .., \delta_{m-1}), \delta_i = r_{i+1} - r_i$ be the differences between the elements of R_o. Then *width* is set to the greatest common divisor of all elements in Δ.

To estimate the validity of this abstraction, we compute a support value of the property. We compute the support value for a single repetition formula as: $\mathit{Support\ value} = \frac{|R_o|}{(\mathit{maxSteps} - \mathit{minSteps}) + 1}$. The support value for a property is the product over all support values of the contained repetition formulas. For example consider the following properties:

```
req[*2] |->##1 ack == 1,  req[*4] |->##1 ack == 1
```

These properties will be combined into the following property:

```
req [*2:4] |-> ack == 1
```

With: $f = (0, 1, 2, 4)$ and $\mathit{Support\ value} = \frac{|(2,4)|}{(4-2)+1} = 0.66$.

E. Splitting properties

We check whether we can increase the average support value by splitting properties using different values for *width*. Let p be a property with support less than 1, thus p must include at least one repetition f with a support value less than 1, i.e., for f not all possible repetition values were observed. Let W be the set including all values for *width* found in the property combination step. We define a subset W_r of W as $W_r = \{w_r | w_r \in W \wedge w_r > 1 \wedge \forall w \in W \backslash \{1, w_r\}, (w_r \mod w) \neq 0\}$. Let width_f be the *width* of f and base_f its base. Further, let $W_d \subseteq W_r$ be the subset of W_r that exactly includes those elements of W_r for which width_f is a divisor ($\forall w_d \in W_r, w_d \mod \mathit{width}_f \equiv 0 \Leftrightarrow w_d \in W_d$).

For each $w_d \in W_d$ we split the property in w_d / width_f parts, where the *width* of each new quadruple is w_d and all the new properties P_n jointly cover exactly all cases covered by the original property. Thus, the *base* of the new quadruple are of the form: $\mathit{base}_f + \mathit{width}_f * n, n < w_d / \mathit{width}_f, n \in \mathbb{N} \cup \{0\}$ and if considering all elements of P_n all possible values of n are hit. Then in each P_n the properties with a support value of 0 are removed. Among all P_n and $\{p\}$ we keep the set with the highest average support and discard the rest.

Example 3. Let us assume we have a set of properties which only differ in the number of repetitions for a single path-condition. Let those numbers be 1, 3, 4, 6, 7, and 9. The combination step will result in repetition quadruple $f = (0, 1, 1, 9)$

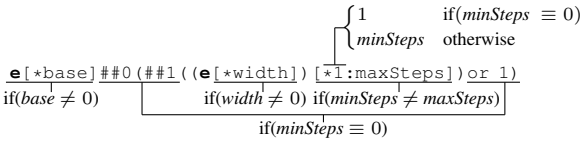

Fig. 3. The representation of an expression e and a repetition quadruple $f = (base, width, minSteps, maxSteps)$ in SystemVerilog.

and the support value will be 0.66. Further, assume in other properties we had observed a *width* of 3. Then the property would be split into three properties with the repetition quadruples: $f_0 = (0, 3, 1, 3)$, $f_1 = (1, 3, 0, 2)$, and $f_2 = (2, 3, 0, 2)$. The property for f_2 will be discarded as its support is 0.0. Both remaining properties will have a support of 1.0. Thus, the average support is increased. Further, as the case for $base = 2$ was never observed it is likely that the property does not hold for $base = 2$.

F. Abstracting repetitions

We abstract from the concrete number of repetitions in the path-constraints. We abstract the repetition by relaxing the requirements for the minimal and maximal number of steps. Given repetition quadruple $f = (base, width, minSteps, maxSteps)$ with $minSteps \neq maxSteps$, the abstractions are:

- *minSteps* is decreased to 1
- *minSteps* is decreased to 0
- *maxSteps* is increased to infinite ($)

Obviously, the resulting abstracted properties could be incorrect. Therefore, in the last step a formal check of the properties is applied.

G. Checking the properties

In the last step we check whether the abstracted properties hold. For this we use a model-checking tool. In case that different abstractions for the repetitions are correct, only the most general property is retained.

H. Representing repetitions

Writing the repetitions in SystemVerilog syntax results in properties often hard to parse for human developers. Figure 3 shows the translation of a symbolic expression e and a repetition quadruple $f = (base, width, minSteps, maxSteps)$ into a SystemVerilog property. Three cases make the property especially hard to read, first having a *base* larger than 0 causes e to be duplicated. Second, the "`##0((##1...)or 1)`" construct is created if *stepWidth* is 0 and finally a *stepwidth* greater 1 causes nested repetition operators. Therefore in such cases we add comments of the form:

```
\\e[*base+n*stepWidth],n>=minStep,n<=maxStep
```

to make the properties easy to understand for a human.

TABLE II
RUNTIME FOR THE DIFFERENT STEPS OF OUR APPROACH

Design	Inst.	Co.+Sim.	Gen.	Check.(all)	Check.(single)
Bridge	13ms	2.7s	50ms	27.2s	1.9s
CPU	42ms	3.2s	3.7s	236m 30s	10m14s

TABLE III
NUMBER OF PROPERTIES FOR THE BUS-BRIDGE

	Initial	Generated	[1:y]	[0:y]	[x:$]	[1:$]	[0:$]
Total	300	51	4	25	25	4	25
Correct	-	45	0	14	16	0	10
Incorrect	-	6	4	11	9	4	15
Final	-	25	0	4	6	0	10

V. EXPERIMENTS

In this section we present the results of applying our approach to a small bus-bridge design and a simple CPU design implementing a subset of the x86 32bit instruction set architecture[1]. All experiments run in a virtual machine having access to 4 cores of an i7-4712MQ and 12GB of RAM. As operation system a 64-Bit Debian 8 was used. Simulation and model checking were done using commercial tools. Table II gives the run-time for the different parts of the approach. The first column gives the name of the design, the second the time for the code transformation and instrumentation, the third the time required for compiling and simulating the instrumented design. The forth column gives the time to generate the initial properties and abstracted properties. All times given in those columns were measured using `std::chrono::system_clock` from the C++-standard library. The last columns give the time required for model checking all properties and the longest time to check a single property. The times in those columns are reported as provided by the model checker. As we can see model checking the properties requires by far the most time.

A. Bus-bridge

The design contains a buffer with parametrized size and data-width. For our case study, we are using a data-width of eight and a buffer size of eight. A user may request to read (ReqR) or write (ReqW) to the bridge. Those operations can be used concurrently. A single request of each type can be pending. If a request is fulfilled, a corresponding acknowledgment (AckR/AckW) signal is asserted. The set of use cases contains two use cases one takes 30 clock cycles and the other 69 clock cycles.

Table III shows the number of generated properties for the different steps of our approach. The column *Initial* gives the number of initially created properties (Section IV.C). The column *Generated* gives the amount of properties generated without the use of abstraction, i.e., the properties after the splitting step described in Section IV.E. Note that for this case study all support values are 1.0, thus splitting was not applied. As we can see there are six incorrect properties, these properties have been generated before the first reset signal in the use cases has

[1] http://www.digitaltechnik.org/examples/Y86_seq.zip

TABLE IV
NUMBER OF PROPERTIES FOR THE CPU DESIGN

	Initial	Generated	[1:y]	[0:y]	[x:$]	[1:$]	[0:$]
Total	2088	171	4	16	38	4	16
Correct	-	58	0	4	22	0	3
Incorrect	-	113	4	12	16	4	13
Final	-	36	0	0	19	0	3

occurred. As the design is not initialized at this point, those incorrect properties are expected. The last five columns contain the results for the different abstractions as described in Section IV.F. Some of the abstractions were not applicable to all properties, for example when *minSteps* was already 1.

The bottom row shows the number of properties included in the final set of properties. As described in Section IV.G we only keep the most abstracted correct version of a property.

The interesting properties generated by our approach can broadly be partitioned into two groups: basic behavior of the design, e.g., if a request cannot be fulfilled directly, the request is stored; valid repetition abstractions of the form [0:$] which describe inputs which can be arbitrarily repeated without changing the internal control-state of the design. This includes idle behavior, but also properties describing constant throughput are generated. Not directly described in the properties but easy to extract is the minimal latency of the design, as we have to search for the lowest parameter of the `$past`-task where the output equals a previous input.

B. CPU design

The CPU is a 32-bit, five stage design with eight general purpose registers, a dedicated program counter, and a status flag to store whether the last result was zero. The status flag is used by the conditional jump instruction. As primary inputs the design has a clock input, a reset input and a data input from memory. Primary outputs are a read enable output, a write enable output, an address output and a data output for controlling the system memory. Further, for debugging purposes, the design has an output providing the opcode of the currently executed instruction. We applied two changes to the CPU design: First, we transformed it into a synchronous design. Second, we added corresponding code to the design such that the simulation ends if the *halt* instruction is executed. The first change was required as our current implementation does not support outputting asynchronous properties. The second change was introduced as the original design requires manually stopping the simulation which does conflict with our goal of an automatic approach.

As use case for this design, we execute the example program shipped with the design. The execution of this program takes 417 clocks cycles, including the clock cycles for resetting the design at the beginning, and 84 instructions are executed, also counting the final *halt* instruction.

Table IV shows an overview of the properties generated for the CPU design. The properties generated for the outputs `bus_RE`, `bus_WE`, and `current_opcode`, completely described the behavior of those outputs once the design is reset. The final set contains between five and six properties for each of those outputs and no property took more than 1 second to be checked.

Further, using the generated properties the structure of the read and write instructions can be extracted, i.e., the opcode of those instructions and the bits indicating read/write from memory; in contrast to read/write from a register. In case of the data output to the memory the generated properties can identify those cycles where the output does not change with respect to the previous cycle, which covers 80% of the output's behavior. However, the properties describing how the value changes, are either incorrect or encode the use case, both cases provide no useful information. For the address outputs our approach yields a similar result. Our approach identifies those clock cycles where the output is constantly 0, which holds 60% of the time, but fails to create meaningful properties for the other cycles. All the useful properties for the data output and the address output were created by the splitting part of our approach. In no other case splitting has been used, i.e., whenever it was used it improved the final result. In total 24 useful properties were created, from which seven require splitting.

C. Comparison with GoldMine

In this section we analyze the differences between our tool and the GoldMine tool [7][2]. As the Bus-Bridge contains Verilog-tasks not supported by GoldMine, the comparison focuses on the CPU design. GoldMine includes a generator for random use cases. For the comparison we use both, the original example program as well as the use case generated by GoldMine. We provided GoldMine with information about the clock and the reset signal. Our approach was provided with information about the clock signal as well. Further, we used the option that GoldMine should target outputs of any bit-width.

The use case randomly generated by GoldMine executes 10,000 clock cycles. Using that use case our approach generates the same 24 useful properties as created with the original use case of 417 clock cycles, as well as a set of very specific properties. The heuristic part of GoldMine generates nine properties from which eight are found to be correct by the model-checker. Four of the correct properties describe the behavior of the write enable output in the stages, where the output is constant. The other four describe the behavior of the read enable output in the stages, where the output is constant. Each of the read enable properties can be directly related to one property created by our approach. For example our approach creates:

```
((rst)
##0((##1((!rst)[*5])[*1:$]) or 1)//(!rst)[*n*5],n>=0
)|->##1 bus_RE ==’b1;
```

and GoldMine creates:

```
(( full[3] == 1 ) |=> ##1 ( bus_RE == 1 ))
```

Note that in this design `full` is a one-hot encoded counter for the current stage of the execution. Where GoldMine describes the output with respect to the internal state, but not how

[2]GoldMine, used by Jan Malburg, was developed by Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign.

that state is reached. Our approach, using the default inputs and outputs, describes how the state is reached but not the internal state. Both cases have their advantages and disadvantages.

We also check how the approaches are affected by different use case sizes. When reducing the length of the use case to 2,700 clock cycles GoldMine no longer creates the properties for the write enable output. The properties for the read-enable output are no longer created by GoldMine once the use case length is reduced below 395 clock cycles. Our approach can create all 24 useful properties even for use cases of 11 clock cycles. This shows that our approach requires far less simulation data to create useful properties. We also check whether the use case might be to small for GoldMine, however increasing the length of the use case to 100,000 clock cycles does not help. Using the example program as use case, causes GoldMine to create the same properties as in case of the use cases with 2,800 or more clock cycles. As the example program, however, only takes 417 clock cycles this is an indication that GoldMine is stronger affected by bad use cases than our approach.

For some clock cycle values, for example 1,300, GoldMine also created 16 properties for the `current_opcode` output. It seems that some internal threshold of GoldMine is hit as the heuristic part creates far more properties for `current_opcode` in those cases, but all except those 16 properties are refuted by the model-checker. Those 16 properties consist of two properties per bit of the output, one property for the high case and one for the low case:

```
(full[4]==1)##1(bus_in[0]==0)|=>(current_opcode[0]==0)
(full[4]==1)##1(bus_in[0]==1)|=>(current_opcode[0]==1)
(full[4]==1)##1(bus_in[1]==0)|=>(current_opcode[1]==0)
\\...
```

All those 16 properties are subsumed by a single property generated by our approach:

```
((rst)
##1((!rst)[*5])[*1:$]
)|->##2 current_opcode==({$past(bus_in,1)}[7:0]);
```

This nicely shows the advantage of word-level properties, as well as allowing more operators in the property.

VI. Conclusion

We presented an approach for automatically generating properties for an HDL-design using DDGs. First, we generate DDGs from a set of use cases for the design. In the next step a set of initial properties is extracted from those DDGs. Through several refinement and abstraction steps a set of general properties is created. In the last step a model-checker is used to verifies the properties.

We evaluated our approach on two designs, the evaluation showed that our approach creates useful properties. Additionally, we compared our approach to the heuristic based dynamic property generation tool GoldMine. Table V gives a short overview of the differences between our approach and GoldMine. The presented approach and GoldMine are using complementary techniques. Our approach uses DDGs to extract relations between signals in the design. This yields word-level properties using a large set of operators. In fact we allow all Verilog operators except the ternary operator, which we translate in a condition and an assignment. Furthermore, our approach requires only a few use cases, as a single observation of a relation is enough to establish the corresponding property. GoldMine on the other hand uses heuristics to establish relations between signals of the design. This allows GoldMine to easily find interesting internal signal which can be utilized in the properties.

TABLE V
Comparison between GoldMine and our approach

	GoldMine	Our Approach
Required sim. clock cycles	many	**few**
Interesting internal signals	**heuristic**	given by user
Asynchronous behavior	**yes**	no
Multi-bit properties	no	**yes**
Expressions in properties	== (0|1)	**All except "?:"**
Equivalence between variables	no	**yes**
Temporal unbounded properties	no	**yes**

Overall GoldMine has an advantage in case of many or unknown internal control signals, where our approach has an advantage in case of few simulation data or complex relation between signals.

References

[1] "2011 Overall Roadmap Technology Characteristics (ORTC) Tables," International Technology Roadmap for Semiconductors, Tech. Rep., 2011.

[2] P. E. Ammann, P. E. Black, and W. Majurski, "Using model checking to generate tests from specifications," in *IEEE International Conference on Formal Engineering Methods*, 1998, pp. 46–54.

[3] G. Fey and R. Drechsler, "Improving simulation-based verification by means of formal methods," in *Asia and South Pacific Design Automation Conference*, 2004, pp. 640–643.

[4] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[5] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2002, pp. 229–239.

[6] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 339–349.

[7] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *Design, Automation and Test in Europe*, 2010, pp. 626–629.

[8] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rülke, "Automatic generation of complex properties for hardware designs," in *Design, Automation and Test in Europe*, 2008, pp. 545–548.

[9] A. DeOrio, A. Bauserman, V. Bertacco, and B. Isaksen, "Inferno: Streamlining verification with inferred semantics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 728–741, 2009.

[10] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for c," in *European Software Engineering Conference*, 2005, pp. 263–272.

[11] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *International Symposium on Automated and Analysis-Driven Debugging*, 2005, pp. 33–42.

[12] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language," *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pp. 1–1315, 2013.