A Guiding Coverage Metric for Formal Verification

Finn Haedicke[1] Daniel Große[1] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{finn, grosse, drechsle}@informatik.uni-bremen.de

***Abstract*—Considerable effort is made to verify the correct functional behavior of circuits and systems. To guarantee the overall success metric-driven verification flows have been developed. In these flows coverage metrics are omnipresent. Well established coverage metrics for simulation-based verification approaches exist. This is however not the case for formal verification where property checking is a major technique to prove the correctness of the implementation.**

In this paper we present a guiding coverage metric for this formal verification setting. Our metric reports a single number describing how much of the circuit behavior is uniquely determined by the properties. In addition, the coverage metric guides the verification engineer to achieve completeness by providing helpful information about missing scenarios. This information comes from a new behavior classification algorithm which determines uncovered behavior classes for a signal and allows to compute the coverage of a signal. To measure the complete circuit behavior we devise a coverage metric for a set of signals. The metric is calculated by partitioning the coverage computation into a safe part and an unsafe part where the latter one is weighted accordingly using recursion. This procedure takes into account that in practice properties refer to internal signals which in turn need to be covered them-self. Overall, our metric allows to track the verification progress in property checking and significantly aid the verification engineers in completing the property set.

I. Introduction

Verification continues to dictate the overall costs of circuit design and the projections see no end to this trend. Hence, to keep up with the progress of fabrication technology the verification approaches require continuous improvements.

Different verification methodologies have been developed over the last decades. Simulation-based validation of assertions, model checking and clever combinations from both worlds are available today (see e.g. [1], [2], [3], [4]). In addition, methods to estimate the verification quality have been proposed. A rich set of methods has been developed for simulation-based verification which essentially identify inadequately exercised portions of the design. An overview on the respective coverage metrics to qualify the testbenches can be found e.g. in [5]. However, these metrics cannot be used for coverage analysis in formal verification because formal methods traverse the underlying *Finite State Machine* (FSM) of the circuit exhaustively. The first coverage methods developed for model checking – which try to answer whether enough properties have been specified – were based on mutations. The general idea is to perform a small modification of the design (or the FSM) and then to check if this is detected by the properties [6]. Many papers followed this concept, e.g. [7], [8], [9], [10], [11], [12], [13]. But still the main problem of these approaches is the sheer number of possible mutations.

Alternative approaches have been devised to check that the property set covers the whole functionality of a design [14], [15], [16], [17]. Conceptually, these approaches analyze whether each output is uniquely determined by the properties. If there is a verification gap, a scenario can be derived that is not yet specified. In this work we use [16] to perform the coverage check. The approach generates a coverage property for each circuit output. The coverage property fails if the properties do not fully determine the output. From the respective counter-example of the *Bounded Model Checking* (BMC) proof an uncovered scenario can be derived. But still the verification engineer wants to know: "*How much of the design behavior have I covered with my properties?*"

Existing approaches are not providing satisfactory answers to this question. A metric based on minterm-counting has been presented in [18] to analyze a property set without any circuit implementation. However, there is no hint which functionality is uncovered and a lot of manual work is necessary to associate the minterms to scenarios. In contrast, [16] allows to generate several specific uncovered scenarios by repeated execution while blocking the previous counter-example. However, such a procedure might become expensive for the following two reasons: First, all possible values of datapath variables used for computing the current output would be enumerated which is infeasible for larger bit widths. Second, a significant manual effort is required to identify the relevant signals of a scenario which control the actual circuit operation. These signals are required to specify the missing properties. Besides this, no metric is available to precisely spot the current status and to track the verification progress.

In this paper, we present a *guiding coverage metric* for formal verification and an algorithm for computing this metric. The core component of our approach is a *new behavior classification algorithm*: For a given signal this algorithm forms a set of classes that describe certain functionality of the RTL circuit. The classes are formed by a modified cone-of-influence analysis and subsequent abstraction, which is executed on the counter-examples from the coverage check. The abstraction determines the common assignment of a class by removing different data values such that only controlling values remain. Then, the *coverage of a signal* can be computed by relating the number of uncovered classes with all possible behavior classes of that signal. However, a coverage metric should provide a measure for the complete circuit. Therefore, we introduce a coverage formulation for a set of signals, typically the circuit's outputs. Since properties for realistic designs often rely on internal signals of the circuit this should also be taken into account automatically. Hence, the proposed metric calculates a *safe coverage* and an *unsafe coverage* which reflects properties without and with dependencies, respectively. Using recursion the latter one is weighted accordingly. If a signal is not fully covered during the coverage computation, the determined uncovered classes give a direct hint which circuit behavior needs to be specified and hence guides the verification engineer.

The remainder of the paper is structured as follows: Section II gives the preliminaries. The behavior classification algorithm is presented in Section III. Section IV introduces the guiding coverage metric and an example. In Section V our approach is evaluated. Finally, the paper is concluded in Section VI.

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project SANITAS under contract no. 01M3088 and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.
978-3-9810801-8-6/DATE12/©2012 EDAA

II. Preliminaries

We assume that the reader is familiar with BMC [19]. In this paper however, the states of the first time frame of the unrolled circuit are not constrained during the SAT check which allows to prove interval properties (for more details see e.g. [20]). For property specification we use a subset of PSL [21].

A basic procedure for the proposed guiding coverage metric is the coverage check presented in [16]. The approach generates a coverage property for each output o of the circuit and reduces the coverage problem to a BMC problem. If this coverage property holds, the value of o is specified by at least one property in any scenario. To perform the coverage check a multiplexer construct is inserted into the circuit and the coverage property basically states that the union of all properties that involve o does not admit behavior else than the one defined by the circuit. Note that this approach can only decide whether an output is uniquely determined by the properties or not (in the second case also a counter-example is provided). But no metric is calculated.

III. Behavior Classification Algorithm

In this section the behavior classification algorithm is presented. The general idea is as follows: The algorithm uses traces (counter-examples) which activate a certain circuit functionality to automatically determine behavior classes of the circuit. Basically, a class is an active path in the circuit and contains the respective controlling signals and values assigned to them. A prerequisite for the proposed classification algorithm is that we can distinguish between control and datapath. For instance, "if" and "case" statements of an HDL description are mapped to multiplexer structures that can be recognized. Similar techniques have been used to scale formal approaches, see e.g. [22], [23], [24].

In the next subsections we present the notation used throughout the paper, the classification algorithm and a simple example demonstrating the classification.

A. Notation and Algorithm

In the remainder of this paper the following general notations are used:

- C denotes a circuit,
- E denotes a set of environment constraints,
- P denotes a set of properties, and
- s denotes a signal / S a set of signals.

The classification algorithm is based on two sub-algorithms: A coverage checking procedure which determines for a given set of properties and a considered signal an uncovered scenario in form of a trace. We use [16] for this task where an uncovered scenario is a counter-example of a failing coverage property. The second sub-algorithm is a *Modified Cone-Of-Influence* (MCOI) procedure: In case of a multiplexer construct, MCOI adds only the data input to the cone which is currently activated by the select input according to the given trace.

Before we present the classification algorithm, two main data structures are introduced:

- *SignalPath*: describes a path of signals, i.e. the signals are connected by standard circuit elements. A signal path is the result of the MCOI procedure.
- *Map SignalPath to Assignment*: stores a mapping from a signal path to a subset of assigned variables of this path. This map is used to represent the behavior classes, the final result of the algorithm.

Algorithm 1 shows the pseudo-code for the classification algorithm. There, CC denotes an instance of the coverage checker. The input of the algorithm consists of the circuit, the environment constraints, the property set, and the signal for which the behavior classes are to be determined. The algorithm starts with the initialization of the coverage checker (line 2) and then performs the main loop. In each iteration, at first a new counter-example (uncovered scenario) is determined by the coverage checker (line 4). Then, the MCOI procedure is executed for this counter-example to extract the signal path for the considered signal. If we have seen this signal path in any previous iteration, a class for this path already exists. Therefore, we can extract the relevant signals and their assignment (line 9). In the next step we abstract from the specific assignment: For each relevant signal, we check if the assignment differs from the corresponding assignment in the current counter-example. If this is the case, we can safely remove this signal since the operation of the current circuit behavior is not controlled by this signal (line 11). Otherwise, if we have not seen the signal path before, all assignments for the contained signals form a new class (line 13 - 14). Now, the updated or new assignments are stored (line 15). Before the algorithm continues the next iteration, all relevant signals of the signal path and their assignments are used to form a blocking clause such that the exact same scenario will not be found again (line 16). At the end, the classification algorithm returns the determined classes.

Algorithm 1: Behavior classification algorithm

```
   Input: Circuit C, Environment constraints E, Property P,
          Signal s
   Output: Set of behavior classes
1  Map SignalPath to Assignment cls ;
2  CC.init(C, P, E, s) ;
3  for n ← 1 to ncex do
4  |  Counterexample cex ← CC.find_uncovered() ;
5  |  if cex was not found then break ;
6  |  SignalPath sp ← MCOI(C, s, cex) ;
7  |  Assignment a ;
8  |  if sp ∈ keys(cls) then
9  |  |  a ← cls[sp] ;
10 |  |  foreach Signal s' ∈ sp do
11 |  |  |  if cex[s'] ≠ a[s'] then remove s' from a ;
12 |  else
13 |  |  foreach Signal s' ∈ sp do
14 |  |  |  a[s'] ← cex[s'] ;
15 |  cls[sp] ← a ;
16 |  CC.add_blocking_clause(a) ;
17 return cls;
```

Now, for a circuit, a set of environment constraints, a given signal, and a property set we can compute the *number of uncovered classes* as

$$\#uncov\_class(C, E, P, s) = |\text{class\_algo}(C, E, P, s)|. \quad (1)$$

We will use this result in the next section to define the coverage for a signal. Before, we provide an example demonstrating the classification algorithm.

B. Example and Observations

To illustrate the classification algorithm, a simple 32-bit ALU as depicted in Figure 1 is used. The circuit either adds or multiplies the inputs a and b, depending on the control input sel. For this example, we assume no properties have been specified (so P is empty). Hence, all 2^{65} input combinations are uncovered cases.

$$\text{metric}(C,E,P,S) = \sum_{s\in S} \frac{\overbrace{\text{cov}(P_s^{\text{Ext}},s)}^{\text{safe coverage}} + \overbrace{\big(\text{cov}(P,s) - \text{cov}(P_s^{\text{Ext}},s)\big)}^{\text{unsafe coverage}} \cdot \overbrace{\text{metric}(P,\text{signals}(P_s^{\text{Int}}))}^{\text{unsafe weight}}}{|S|} \tag{2}$$

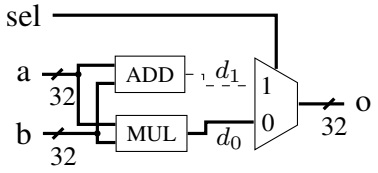

Fig. 1. A simple ALU

In the first iteration of the classification algorithm, the coverage checker computed the counter-example $a = 1 \wedge b = 3 \wedge sel = 0$, which results in an multiplication with the result $o = 3$. Then, the algorithm executes MCOI for this counter-example. The resulting active signal path is depicted as bold lines in Figure 1. At the multiplexer, the cone is restricted to the select input and the input activated by the counter-example. The resulting path defines a new class and the assignment of the active signals is stored ($a = 1, b = 3, sel = 0, d_0 = 3, o = 3$). From this assignment, a blocking clause is created, so that this scenario can not occur again.

Assuming that the next counter-example also triggers the multiplication, e.g. $a = 1 \wedge b = 4 \wedge sel = 0$, the same active path is calculated and the assignment for this class is updated. In this case b, d_0 and o differ and are therefore dropped from the class assignment, which is now $a = 1$ and $sel = 0$. This actual assignment is used to build the blocking clause and hence excludes any combination of b with these values. With a final counter-example for this class, also a will be dropped and only $sel \neq 0$ will be used as blocking clause. Hence, no more multiplication counter-examples will be generated and this class is completed with the signal path and the controlling assignment $sel = 0$.

The addition operation can also be classified with a maximum of 3 counter-examples, so altogether only 6 uncovered counter-examples had to be generated to completely classify the behavior of this ALU.

In general, no more counter-examples than the number of inputs in the MCOI need to be generated for a class to be completely classified. The order in which the counter-examples are generated is not relevant as the classes are only identified via their respective active signal path.

Please recall that the proposed classification algorithm requires multiplexer elements to be explicitly available in the circuit representation. But this is not a practical limitation since our approach can be applied before logic optimization. The resulting complete property set can be used for any circuit representation.

IV. Guiding Coverage Metric

This section introduces the proposed guiding coverage metric. At first, we describe how to compute the coverage of a signal. Using this result, the recursive formulation of our coverage metric is given. We illustrate the metric by means of an example. Finally, the workflow to achieve full coverage using the proposed metric is presented.

A. Coverage of a Signal

Based on the behavior classification algorithm as introduced in the previous section we can compute the coverage of a signal as follows:

$$\text{cov}(C,E,P,s) = 1 - \frac{\#uncov\_class(C,E,P,s)}{\#uncov\_class(C,E,\emptyset,s)} \tag{3}$$

The denominator in the fraction uses no properties (P is empty) and hence the number of all behavior classes for s is determined. The numerator represents the number of classes which are not covered. Overall, complementing this fraction gives the coverage of s by P. Furthermore, it is easy to see if a class is covered (by a single property or in combination through several properties), this class is not found by the classification algorithm anymore since no counter-examples exist. Therefore, this class does not appear in the numerator. In the best case, if the signal is completely described by the properties, the numerator becomes 0 and hence the overall result becomes 1.

In the following, we introduce our coverage metric based on this result.

B. Coverage Metric

Our coverage metric meets two practically very important requirements: First, a single number is computed for a set of signals (typically the set of circuit outputs is used). Second, our metric takes into account if properties use internal signals and ensures that theses signals also have to be covered since otherwise the overall coverage cannot become one.

The following notations are required for our metric:

- P^{Int}: refers to a set of properties with dependencies on internal signals,
- P^{Ext}: refers to a set of properties without dependencies on internal signals, and
- P_s: denotes a set of properties specifying the signal s.

Equation (2) gives the recursive formulation of our coverage metric.[1] The metric is computed for the circuit C, the environment constraints E, the property set P and the set of signals S. For each signal s from S its resulting coverage is determined and added up. The coverage for s is calculated on the basis of safe coverage, unsafe coverage, and unsafe weight. *Safe coverage* accounts for the coverage of the signal s where only external properties determine the value of s, i.e. no internal signals are used to constrain the behavior of the current signal (neither in the antecedent nor in the consequent of the property). Thus, this number can be used without additional justification. In contrast, *unsafe coverage* (first part of the second addend) refers to the coverage fraction where dependencies on internal signals exist. The unsafe coverage is computed as overall coverage minus safe coverage but weighted accordingly (see *unsafe weight*). To determine the weight, the metric is evaluated again, but now instead of the signals S we use all internal signals occurring in the properties (denoted as $\text{signals}(P_s^{\text{Int}})$). As a result, we ensure whether a new property is specified for a certain signal and this property uses an internal signal, this internal signal itself needs to be covered. If this is not done, the respective weight can not become 1 and hence the overall coverage can not become 1.

Due to the structure of the properties it might happen that the dependencies form a *loop*, e.g. the coverage of a signal s directly or indirectly depends on a signal t which again

[1] Note that in $cov(\ldots)$ C and E are omitted for readability.

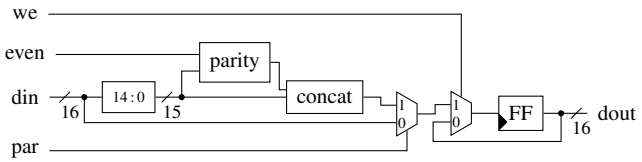

Fig. 2. 16 bit memory element with optional parity generation

depends on s. In such a case the coverage of s would be used to calculate the weight for s. Therefore, the plain coverage $\mathrm{cov}(C, E, P, s)$ (see (3)) is returned if the calculation detects a loop. The plain coverage is an upper bound since this is the maximum possible coverage for s given P (ignoring dependencies). This upper bound is now used to calculate the coverage of t and finally for the weighted coverage of s. This is correct because full coverage can only be obtained if this upper bound is 1 and also all signals along the loop are fully covered. If s or any signal in the loop is not completely covered, the metric will detect and reflect this with a value less then 1.

Altogether, the recursive formulation guarantees that the verification engineer covers all depending signals by reducing these signals finally to inputs or covered states. As described, the recursive coverage computation is necessary for the depending signals. Since the metric is based on the circuit classification algorithm to determine the uncovered circuit behavior classes, control signals and their respective assignments are calculated and available. This allows to easily specify the remaining properties and to achieve full coverage. In that sense the verification engineer is guided during the property specification process with target-oriented information. However, it is important to note that the determined classes should only be used as a reference point for property specification. Creating a property one-to-one from a class may leads to weak properties which are too design dependent.

C. Example

We illustrate our guiding coverage metric for a 16 bit memory element with optional parity generation (see Figure 2). If parity is enabled (input *par*) and data is written into the memory (by input *din*), the MSB of the data is replaced by the parity calculated from the data bits 0 to 14. The parity computation can be controlled to be odd or even via the input *even*.

When running the classification algorithm for the output $dout$ with no environment constraints and the empty property set, i.e. executing $uncov\_class(C, \emptyset, \emptyset, dout)$, four classes are found:

1) No data is written: $we = 0$
2) Plain data is written: $we = 1 \wedge par = 0$
3) Odd parity data is written: $we = 1 \wedge par = 1 \wedge even = 0$
4) Even parity data is written: $we = 1 \wedge par = 1 \wedge even = 1$

A naive iteration of the coverage check for $dout$ using the empty property and blocking each returned counter-example would generate 2^{54} counter-examples in total (3+16 bit inputs over two cycles plus 16 bit for the state).

For the circuit at hand, the three properties shown in Figure 3 have been specified. Performing a coverage check for $dout$ using [16] gives the result that no uncovered scenarios exist. However, applying the developed metric the result is only a coverage of 50% for $dout$ as can be seen in the upper part of Table I. The table gives the necessary information to follow the steps of the recursive calculation. The first column gives the current set of signals, while the next two columns provide the external and internal properties for the actual signal s, respectively. The next three columns give

```
property pWriteP =
always (
  we == 1
  && par == 1
) -> (
  next(dout) == (parity, din[14:0])
);

property pWriteW =
always (
  we == 1
  && par == 0
) -> (
  next(dout) == din
);

property pNoWrite =
always (
  we == 0
) -> (
  next(dout) == dout
);
```

Fig. 3. Initial property set P_I for memory element

TABLE I
COVERAGE METRIC FOR MEMORY ELEMENT

S	P_s^{Ext}	P_s^{Int}	safe	unsafe	unsafe weight	depends	$\sum$
$dout$	pWriteW, pNoWrite	pWriteP	50%	50%	0	parity	50%
$parity$	$\emptyset$	$\emptyset$	0%	0%	0	–	0%
$dout$	pWriteW, pNoWrite	pWriteP	50%	50%	0.5	parity	75%
$parity$	pParityOdd	$\emptyset$	50%	0%	0	–	50%
$dout$	pWriteW, pNoWrite	pWriteP	50%	50%	1	parity	100%
$parity$	pParityOdd, pParityEven	$\emptyset$	100%	0%	0	–	100%

the percentage of save coverage, unsafe coverage, and unsafe weight, respectively. The column *depends* lists the depending signals (if there are any) and the last column gives the overall added result. As can be seen only 50% of the behavior of *dout* is covered by the properties (last number in first row). The safe coverage of *dout* is 50% since class 1 and class 2 (see enumeration above) are covered by the properties *pNoWrite* and *pWriteW*, respectively. As unsafe coverage we get 50% ($100\% - 50\%$). Since the property *pWriteP*, which covers class 3 and class 4, uses the internal signal *parity* (see consequent of the property) the metric is recursively applied for this signal which is obviously not determined by the given properties. Thus, the safe and unsafe coverage of *parity* is 0%, no depending signals exists and hence the unsafe weight for $dout$ in the first row becomes 0. This explains the overall result of 50%.

Besides the coverage value for $dout$ also the dependency on $parity$ including the classification for this signal is returned by our approach. The uncovered classes resulting from the classification of $parity$ are summarized in Table II. As can be seen there are two uncovered classes. For the first class the active path contains the signals relevant for the odd

TABLE II
UNCOVERED CLASSES OF $parity$

Class	Active path	Assignment
1	din, even, lower_din, parity, parity_odd	$even = 0$
2	din, even, lower_din, parity, parity_even	$even = 1$

```
property pParityOdd =
always (
  even == 0
) -> (
  parity == (^din[0..14]) // XOR reduce
);
property pParityEven =
always (
  even == 1
) -> (
  parity == !(^din[0..14]) // XOR reduce
);
```

Fig. 4. Properties for parity computation

parity computation and the final assignment that remains after intersection of the counter-examples for this path only contains the signal $even$ set to 0. In case of the second class obviously the even parity mode is found. By step-wise adding the respective properties (see Figure 4) we get results shown in the middle and bottom of Table I, respectively. With the additional property $pParityOdd$ we achieve a coverage of 75%. By including both properties 100% coverage results.

D. Full Coverage Workflow

This subsection describes the workflow to achieve full coverage using the proposed metric. The steps to follow are:

1) Define the set of signals S to be analyzed. Typically S contains the circuit outputs.
2) Apply the proposed metric.
3) Select a signal $s \in S$ which is not fully covered.
4) If the sum of safe coverage and unsafe coverage for s is less than 100%, properties describing s are needed. For closing the gap(s), consider the uncovered behavior classes as determined during the calculation of the coverage metric. Formulate one or more properties according to the specification which cover at least one behavior class. If possible avoid the creation of new dependencies by only relying on inputs or signals already in use.
5) If the sum of safe coverage and unsafe coverage for s is 100% (recall s is not fully covered so the unsafe weight has to be less than 1), proceed to fully cover the dependent signals.
6) Continue with Step 2 until all signals in S are covered.

Following these steps, there are degrees of freedom how to continue the verification process. If coverage gaps for depending signals are found and a property is added to close this gap, one may proceed to verify new potential depending signals. This would correspond to a depth-first verification procedure. In contrast, repeating step 4 gives a breath-first verification procedure. In general, internal signals should only be used in properties if there is no alternative.

V. Evaluation

For an in-depth evaluation of the proposed guiding coverage metric a *Memory Management Unit* (MMU) has been chosen. The interface of the design is depicted in Figure 5.

The MMU has a CPU connection (left side) as well as a memory connection (right side) and can buffer a single write request.

During the development of the MMU bounded model checking has been employed to verify the correctness of the implementation. Thus, a property set for the MMU already existed. Using this property set an overall coverage of 42.3% for the 7 outputs of the MMU was calculated. While computing the coverage metric 4 dependent internal signals used by the properties have been found. The results of our metric for all

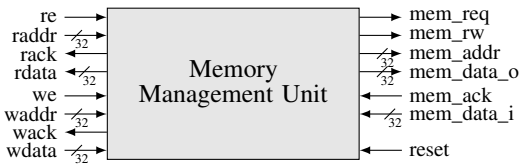

Fig. 5. Memory management unit

TABLE III
Result of the coverage metric for the initial property set

signal	safe	unsafe	unsafe weight	$\sum$
mem_addr	66.7%	12.8%	0.472	72.7%
mem_data_o	0.0%	67.4%	0.535	36.0%
mem_req	85.7%	8.9%	0.502	90.2%
mem_rw	84.5%	3.4%	0.514	86.3%
rack	0.0%	0.0%	0.294	0.0%
rdata	0.0%	0.0%	0.294	0.0%
wack	0.0%	25.0%	0.441	11.0%
state	50.0%	12.5%	0.558	57.0%
wbuf_addr	15.0%	0.0%	0.362	15.0%
wbuf_data	15.0%	0.0%	0.362	15.0%
wbuf_full	31.2%	0.0%	0.308	31.2%

these signals are listed in Table III. In the upper part of the table the outputs signals are shown while the lower part provides the data for the 4 internal signals. Column *signal* specifies the name of the signal. The next three columns provide safe coverage, unsafe coverage, and unsafe weight. The last column shows the coverage value for the current signal. As can be seen there are several signals which have a high coverage, but there are also cases where the coverage is 0%. We first consider one of the two signals which have a coverage of 0% in more detail. Even if the achieved coverage of $rack$ is 0%, one property has been specified for $rack$ (see Figure 6). However, this property only verifies a very small portion of $rack$'s behavior. In terms of our coverage metric this means not even a single class is covered by the $READ\_return$ property. This becomes evident when looking at the uncovered classes of $rack$ as determined during the computation of our metric. They are shown in Table IV. When considering the assignments in class 2 (this class contains the $READ$ state which is the same scenario as considered in the property $READ\_return$), it can be seen that the signals $rack$ and mem_ack are both zero. This complements the specified value of $rack$ in the

```
property READ_return =
always(
  !reset && state == READ && mem_ack
) -> (
  rack && rdata == mem_data_i
);
```

Fig. 6. Only property for $rack$

TABLE IV
Uncovered classes of $rack$ using property $READ\_return$

Class	Assignment
1	$reset \wedge \neg rack$
2	$\neg reset \wedge state = READ \wedge \neg mem\_ack \wedge \neg rack$
3	$\neg reset \wedge state = IDLE \wedge \neg wbuf\_read \wedge \neg rack$
4	$\neg reset \wedge state = IDLE \wedge wbuf\_read \wedge rack$
5	$\neg reset \wedge state = WRITE \wedge \neg wbuf\_read \wedge \neg rack$
6	$\neg reset \wedge state = WRITE \wedge wbuf\_read \wedge rack$
7	$\neg reset \wedge state = READ\_WRITE\_PENDING$

From the assignments (right column) the buffered internal signals have been removed to increase readability.

```
property RACK_WRITE_IDLE =
always (
  !reset && (state == WRITE || state == IDLE)
) -> (
  rack == wbuf_read
);
```

Fig. 7. Additional property for $rack$

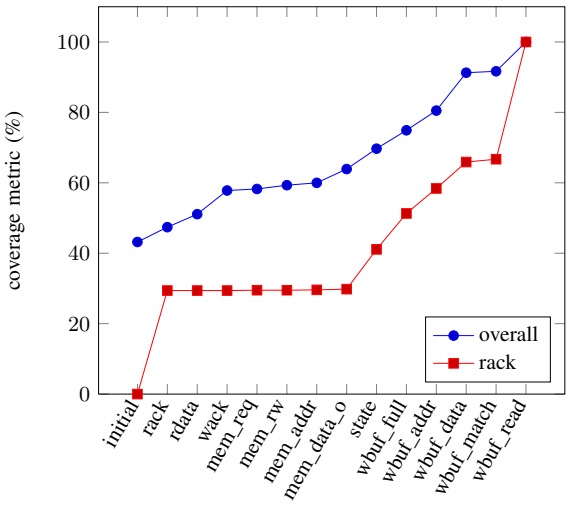

Fig. 8. Coverage evolution when adding properties using the guiding coverage metric

consequent of the property: Altogether, mem_ack is assigned to $rack$ in state $READ$ if no reset is requested. In other words, the existing property only describes "half" of the READ state behavior class. To summarize, since the property $READ\_return$ describes only a subset of a class, the class is not covered and hence the coverage of $rack$ becomes 0%.

For the other classes of $rack$ additional properties need to be formulated. Class 1 refers to the reset behavior and therefore the existing reset property has been extended to include $rack = 0$. The classes 3 to 6 can be summarized in a single property as shown in Figure 7. In both states, i.e. in *IDLE* and *WRITE*, $wbuf\_read$ is assigned to $rack$. Two classes for both states are identified by our classification algorithm, class 3 and 4 for *IDLE* and class 5 and 6 for *WRITE*. There are two classes in both cases because an if-condition is used in the circuit to check the value of $wbuf\_read$ and therefore different circuit paths become active enabling the then-block or the default assignment. The last class does not specify a value for $rack$, which indicates that the value is not explicitly set here, but depends on the value of another signal on the actual active signal path (it is mem_ack). After updating the property set, a total coverage of 45.5% resulted. Thereby, the coverage of $rack$ increased to 29%. 100% for $rack$ was not achieved since new dependencies to internal signals emerged.

We continued to fully cover the remaining signals. Therefore, we covered all the outputs in the next steps, i.e. we wrote properties using external signals (inputs or proven states) as long as possible and then finalizing the remaining behavior by relying on internal signals. If new signal dependencies have been introduced (and hence potentially unsafe coverage) we postponed the verification of these signals. Figure 8 shows the evolution of the overall coverage and the coverage for $rack$ when following this procedure. The x-axis denotes the chosen signal to be fully described by (additional) properties. As can be seen verifying the other outputs has no effect on the coverage of $rack$ but the overall coverage increases. As soon as the dependencies of $rack$ are covered ($state$ to $wbuf\_read$) the coverage of $rack$ increases. The signals $wbuf\_match$ and $wbuf\_read$ are new internal signals relevant for the behavior of $rack$. In particular, $wbuf\_read$ has a strong influence on $rack$ as the rapid growth of $rack$'s coverage from $wbuf\_match$ to $wbuf\_read$ shows. The signal $wbuf\_read$ is used in 4 out of 7 classes.

During the evaluation we observed that the run-time for computing the coverage metric is on average about 5 to 10 times higher than proving the properties.

In total, with the proposed coverage metric we first determined the verification status. Then, we successfully tracked the verification progress while completing the property set. In doing so the uncovered class information always provided very valuable information.

VI. Conclusions

In this paper we have presented a coverage metric for formal verification. The metric computes a single number given a circuit, environment constraints, a property set and the set of signals to be covered. The metric takes dependencies on internal signals into account which are typically found in realistic property sets. Therefore, the metric is automatically applied recursively. In addition, the verification engineer is guided towards full coverage since during the coverage computation uncovered behavior classes are determined. These classes are very helpful to formulate the missing properties.

References

[1] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Kluwer Academic Publishers, 2003.
[2] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.
[3] M. Ganai and A. Gupta, *SAT-Based Scalable Formal Verification Solutions (Series on Integrated Circuits and Systems)*. Springer, 2007.
[4] V. Bertacco, *Scalable Hardware Verification with Symbolic Simulation*. Springer, 2006.
[5] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design and Test of Computers*, vol. 18, no. 4, pp. 36–45, 2001.
[6] Y. Hoskote, T. Kam, P. Ho, and X. Zhao, "Coverage estimation for symbolic model checking," in *DAC*, 1999, pp. 300–305.
[7] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage metrics for temporal logic model checking," in *Tools and algorithms for the construction and analysis of systems*, ser. LNCS, no. 2031, 2001, pp. 528 – 542.
[8] H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi, "A practical approach to coverage in model checking," in *CAV*, ser. LNCS, vol. 2102. Springer Verlag, 2001, pp. 66–77.
[9] N. Jayakumar, M. Purandare, and F. Somenzi, "Dos and don'ts of CTL state coverage estimation," in *DAC*, 2003, pp. 292–295.
[10] O. Kupferman, "Sanity checks in formal verification," in *In Proc. of CONCUR*, 2006, pp. 37–51.
[11] A. Fedeli, F. Fummi, and G. Pravadelli, "Properties incompleteness evaluation by functional verification," *IEEE Trans. on Comp.*, vol. 56, no. 4, pp. 528–544, 2007.
[12] O. Kupferman, W. Li, and S. A. Seshia, "A theory of mutations with applications to vacuity, coverage, and fault tolerance," in *FMCAD*, 2008, pp. 1–9.
[13] H. Chockler, D. Kroening, and M. Purandare, "Coverage in interpolation-based model checking," in *DAC*, 2010, pp. 182–187.
[14] K. Claessen, "A coverage analysis for safety property lists," in *FMCAD*, 2007, pp. 139–145.
[15] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, "Complete formal verification of Tricore2 and other processors," in *Design and Verification Conference (DVCon)*, 2007.
[16] D. Große, U. Kühne, and R. Drechsler, "Analyzing functional coverage in bounded model checking," *IEEE Trans. on CAD*, vol. 27, no. 7, pp. 1305–1314, 2008.
[17] J. Bormann, "Vollständige funktionale Verifikation," Ph.D. dissertation, Technische Universität Kaiserslautern, 2009.
[18] M. Oberkönig, M. Schickel, and H. Eveking, "A quantitative completeness analysis for property-sets," in *FMCAD*, 2007, pp. 158–161.
[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS*, 1999, pp. 193–207.
[20] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *IEEE Trans. on CAD*, vol. 27, no. 11, pp. 2068–2082, 2008.
[21] *Accellera Property Specification Language Reference Manual, version 1.1*, http://www.pslsugar.org, 2005.
[22] R. Hojati and R. K. Brayton, "Automatic datapath abstraction in hardware systems." in *CAV*, 1995, pp. 98–113.
[23] G. Kamhi, O. Weissberg, and L. Fix, "Automatic datapath extraction for efficient usage of HDD," in *CAV*, ser. LNCS, vol. 1254. Springer Verlag, 1997, pp. 95–106.
[24] Z. S. Andraus and K. A. Sakallah, "Automatic abstraction and verification of verilog models," in *DAC*, 2004, pp. 218–223.