Property Analysis and Design Understanding in a Quality-Driven Bounded Model Checking Flow

Ulrich Kühne
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
ulrichk@informatik.uni-bremen.de

Daniel Große
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
grosse@informatik.uni-bremen.de

Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
drechsle@informatik.uni-bremen.de

***Abstract*—In the design process of digital systems, functional verification is a major issue. Generally, formal methods like bounded model checking (BMC) offer the highest quality of the verification results, especially when used in combination with techniques that check if a set of properties forms a complete specification of a design. However, in contrast to simulation-based methods, like random testing, formal verification requires a detailed knowledge of the design implementation. Formalizing a specification as a set of properties is a tedious and time consuming process. In this paper, we show the application of techniques to aid the verification engineer in writing properties in a quality-driven BMC flow, that have been introduced in [1]. The first method can be used to remove redundant assumptions from properties and to separate different scenarios. The second technique, here called inverse property checking, takes an expected behavior of a design and automatically generates valid properties that can be checked for conformance with a specification. Both techniques can serve to reduce the number of iterations to obtain full coverage, when integrated with the verification flow. The benefits of the techniques are demonstrated with a memory management unit.**

I. Introduction

With an increasing design size, automation and tool support of the hardware verification process is indispensable. Concerning the verification methodology, there are mainly two different paradigms – simulation-based verification and formal verification. Simulation-based approaches rely on a test bench that should capture all relevant scenarios. The simulation results are compared to a golden reference model. In formal verification the functional behavior is described by properties which are checked on the design using symbolic techniques [2]. Although simulation-based verification is still widely used in industry, formal verification generally offers the highest quality.

While for simulation based methods, various coverage metrics – like code coverage or branching coverage – are used to measure the quality of simulation, these metrics cannot be directly ported to formal verification. However, there are several techniques to ensure that a set of properties covers the whole functionality of a design [3], [4], [5], thus guaranteeing that the written properties form a complete specification. Although this improves the formal verifcation flow and provides a well-defined stopping criterion, the manual effort for formal verification is still high. While the setup for random simulation is quite easy as it treats the *design under verification* (DUV) as a black box, more sophisticated techniques like directed tests require some insight into the implementation. Finally, for writing a high-quality property set that formally captures the specification, deep knowledge of the DUV internals is necessary. Therefore, the time needed to get the formal verification up and running is significantly higher. In contrast, once the property suite is complete, the task of verification is fulfilled, while simulation-based methods may never come to meet the coverage requirements in a reasonable time [6].

As a conclusion, it is inevitable for the verification engineer to achieve a good design understanding in order to match the specification with the implementation. This may take a significant amount of time and a meticulous inspection of the specification and the RTL code. Making it even worse, in many cases the specification will be incomplete or outdated. The same applies for source code documentation. Thus, to achieve a shorter ramp-up time for formal verification, it is necessary to provide support for the verification engineer to guide the verification process.

In order to decrease the overhead of formal methods, techniques to aid the verification engineer in design understanding and to ease the formalization of the specification have been introduced in [1]. The first method automatically analyzes a given property, identifying too strong constraints on the environment or the internal state of the DUV. Using this technique, the number of iterations to achieve full coverage can be reduced. Furthermore, the technique provides a true gain in design understanding by revealing which parts of the assumptions are sufficient to prove a property. Based on this approach, the second technique automatically generates properties, given an expected behavior as a temporal expression. With this *inverse property checking*, the user can interactively query the design to find out how the abstract concepts of the specification are implemented. The generated properties can then be inspected and verified with the specification.

In this paper, the integration of these techniques with a formal coverage analysis are demonstrated in an experi-

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerisoftXT under contract no. 01 IS 07008 C.

mental case study. Together with a coverage analysis, the formalization of a specification can be approached from both ends: making the written properties as concise as possible by analyzing them and revealing new uncovered behavior and integrating this behavior in the property suite using inverse property checking.

The paper is structured as follows. In the next section we will provide the basics on the used verification and coverage techniques. The techniques mentioned above are briefly reviewed in Sections III and IV, followed by the case study in Section VI. The work is concluded in Section VII. A discussion of the presented techniques and of related work can be found in [1].

II. Preliminaries

A. Verification Setting

Bounded Model Checking (BMC) has been introduced in [7]. In contrast to the original BMC, we use an *all states verification*, meaning that the properties are proven for arbitrary starting states [8], [9].

Formally, for a design with its transition relation T_δ, a BMC instance for a property p over the finite time interval $[0, c]$ is given by

$$\bigwedge_{i=0}^{c-1} T_\delta(s_i, s_{i+1}) \;\wedge \neg\, p\,, \tag{1}$$

where p may depend on the inputs, states and outputs of the circuit in the time interval $[0, c]$. This verification problem can be formulated as a *Boolean satisfiability* (SAT) problem by unrolling the circuit for c time frames and generating logic for the property. A satisfying assignment corresponds to a case where the property fails – a counter-example. Allowing arbitrary starting states may lead to false negatives, i.e. counter-examples that start from an unreachable state. In such a case these states are excluded by additional assumptions in the property or assuming proven invariants. But, for BMC as used here, it is not necessary to determine the diameter of the underlying sequential circuit. Thus, if the SAT instance is unsatisfiable, the property holds.

For the formulation of the properties, we use a subset of PSL (property specification language [10]). A property has the form of an implication $A \to C$. Here, the antecedent A contains a conjunction of assumptions on the state and inputs of the design, like e.g. environment constraints or a specific configuration setting. The consequent C then describes the intended behavior. The used operators are the typical HDL operators like logic, arithmetic and relational operators. The timing is expressed using the operators $next$ and $prev$.

B. Coverage Analysis

To achieve a high quality verification result, the properties must cover the entire behavior of the DUV. As non-trivial verification scenarios have to be considered and properties may get quite complex, this achievement is not obvious to the verification engineer. Instead, a formal check can be carried out to prove that the functionality of the DUV is fully covered by the properties, as described in [5]. There, it is checked for each output of a hardware module whether a set of properties uniquely determines the value of the output for each possible scenario of states and inputs. If the check fails, an uncovered scenario in form of a counter-example is presented to the user. Full coverage in terms of this approach means that a signal is determined by a set of properties for all possible state and input scenarios.

Alternative approaches for analyzing coverage for formal property verification can be found in [3], [4].

III. Property Analysis

The most common cause for the coverage check to fail are too strong assumptions in the properties. For this reason, it is desirable to have an automatic support for the user in writing properties as concise as possible, saving time for further iterations on the way to full coverage.

The idea of the property analysis approach is to iteratively find subsets of essential subexpressions of the antecedent. A subset is essential if removing the contained subexpressions invalidates the property. In this way, all possible combinations of subexpressions can be constructed that are sufficient to guarantee the validity of the property. The overall flow of the property analysis is depicted in Algorithm 1. The steps are described in the following.

Given a property in the form $A \to C$, in order to analyze the antecedent, it is decomposed into its subexpressions. For each subexpression, a free Boolean variable d_i (*disable*) is introduced to control the disabling of the expression in the antecedent. In this way, an antecedent $A = A_1 \wedge A_2 \wedge \cdots \wedge A_n$ is transformed to

$$A' = \bigwedge_{i=1}^{n} (d_i \vee A_i). \tag{2}$$

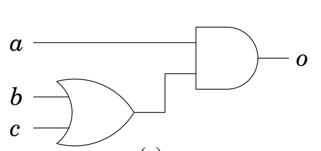
In the next step, all combinations of disabled subexpressions are extracted that falsify the overall property. This is done iteratively by solving the SAT instances Q_k for k ranging from 1 to n, the number of subexpressions:

$$Q_k = \bigwedge_{i=1}^{n} (d_i \vee A_i) \;\wedge\; (\sum d_i = k) \;\wedge\; \neg C \tag{3}$$

The design is unrolled within the involved time interval (omitted in Equation (3)). The found assignments to the d_i variables are stored in a BDD $\mathcal{E}$ (*Binary Decision Diagram* [11]) and a blocking clause is added to the instance in order to conduct the solver to the next solution. If an assignment $a : (d_1, d_2, \ldots, d_n) \mapsto \{0, 1\}^n$ is found, only the cube consisting of the positive literals $a(d_i) = 1$ is stored, representing all supersets of the subset of disabled subexpressions. As disabling the identified subset already falsifies the property, disabling more subexpressions will falsify it as well. Thus, we do not need to search for these supersets any more. By starting with $k = 1$ and incrementing it, we are able to block the most general supersets first and thereby reduce the number of solver

Algorithm 1: propertyAnalysis

```
Input: circuit $M$, property $P = (A \rightarrow C)$
1 $A'$ = reformulated antecedent
2 for $(k = 1 \ldots n)$ do
3     repeat
4         find assignment for $Q_k$
5         store cube of $d_i$ in BDD $\mathcal{E}$
6         block assignment
7     until UNSAT ;
8 compute BDD $\mathcal{S} = \neg\mathcal{E}$
9 compute set cover of $\mathcal{S}'$
```



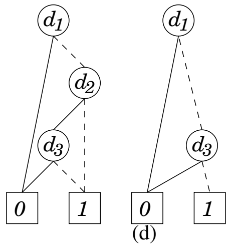
(a)

d_1	d_2	d_3	k
1	0	0	1
0	1	1	2
-	-	-	3

(b)

```
property P =
  always(
    a == 1 &&
    b == 1 &&
    c == 1 &&
  ) -> (
    o == 1
  );
```

(c)



(d)

Fig. 1. Example for Algorithm 1

calls. The corresponding blocking clause for assignment a is $\bigvee\limits_{a(d_i)=1} (\neg d_i)$.

If for a given k no more solutions can be found, k is incremented until it reaches the number of subexpressions n. At the end, $\mathcal{E}$ contains all subsets of assumptions that falsify the property. By calculating the BDD $\mathcal{S} = \neg\mathcal{E}$, we obtain a collection of all subsets of assumptions that preserve the validity of the property. Note that negation on BDDs is a constant time operation when using complement edges.

The resulting sufficient antecedents can then be computed as a set cover of the subsets in $\mathcal{S}$. This is done by iteratively removing cubes from the BDD until the zero function is obtained. By preferring the paths with the least number of low edges, antecedents with a small number of activated subexpressions are picked first. These are usually the most interesting results for the user.

Example 1: Consider the circuit shown in Figure 1(a), implementing the function $o = a \wedge (b \vee c)$. The naive property in Figure 1(c) states that o is 1 whenever all inputs are 1. Starting the analysis, for $Q_1 = (d_1 \vee a = 1) \wedge (d_2 \vee b = 1) \wedge (d_3 \vee c = 1) \wedge (\sum d_i = 1) \wedge \neg(o = 1)$, we find the single solution $(d_1, \neg d_2, \neg d_3)$, as shown in the first row of the table in Figure 1(b). This means that disabling

Algorithm 2: inversePropertyCheck

```
Input: circuit $M$, expected behavior $e$
1 compute maximum delay $d_{max}$
2 compute witness $a$ of length $d_{max} + 1$
3 compute set $IS$ of influencing signals
4 build initial property $P = (\bigwedge_{(s,t) \in IS} (s^t = a(s,t))\ ) \rightarrow e$
5 apply propertyAnalysis on $P$
```

$A_1 = (a = 1)$ invalidates the property. The only solution for Q_2 is $(\neg d_1, d_2, d_3)$. There is no more solution for $k = 3$ that does not include the already found subsets. The resulting BDD $\mathcal{S} = \neg\mathcal{E}$ is shown in Figure 1(d) on the left (low edges are represented by dashed lines). From this, we pick and remove the cubes corresponding to the paths to the terminal 1-node. The sufficient antecedents to satisfy the consequent $(c = 1)$ are then $(a = 1)\&\&(b = 1)$ and $(a = 1)\&\&(c = 1)$, corresponding to the paths $(\neg d_1, \neg d_2)$ and $(\neg d_1, \neg d_3)$, respectively. (The BDD on the right of Figure 1(d) represents the intermediate result after removing the first solution $(a = 1)\&\&(b = 1)$.) $\square$

Note that the sufficient antecedents can also be constructed by computing all *minimal unsatisfiable subformulas* [12] of the inital property. This relation is further discussed in [1].

In summary, the property analysis removes unnecessary assumptions of a concrete verification scenario on the one hand. On the other hand different scenarios can be identified and separated.

IV. Inverse Property Checking

For a given property, the above analysis can be used to extract all the essential assumptions within the antecedent. Instead, based on the analysis, the user can start with a given expected behavior or proof goal for which legal antecedents are constructed automatically. The resulting properties hold by construction and can then be inspected by the user to find out if they comply with the specification. This approach is denoted as *inverse property checking*. It reverses the normal process of finding a formal description for a given functionality, where the verification engineer tries to extract the correct setting from the specification and the RTL implementation. Here a valid description of the implementation is extracted automatically, that can then be compared to the specification.

The idea is to find a witness for a given proof goal. A witness in this case is a complete assignment to the signals of the DUV, which also satisfies the given consequent expression. The witness is then generalized using the property analysis. In this way, it is revealed which of the assignments in the witness are sufficient for a given proof goal. A sketch of the inverse property checking is shown in Algorithm 2.

The expected behavior for inverse property checking is specified as a PSL expression e. In order to construct a witness for the expected behavior, a BMC instance is build by unrolling the circuit and adding the constraints for e. Therefore, for the circuit the maximum delay d_{max} between any of the inputs or state signals and the signals involved in e

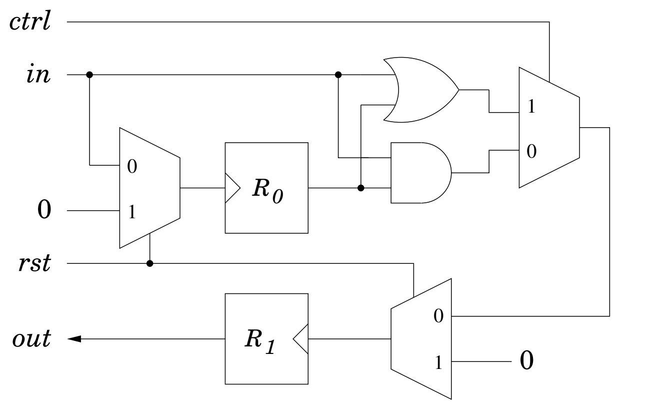


Fig. 2. Simpe logic unit

is calculated by a depth-first search on the circuit. The circuit is then unrolled in the interval $[0, d_{max}]$.

If the instance is unsatisfiable, the expected behavior e can never be observed in the design. Otherwise, we obtain a witness $a : S \times T \rightarrow \mathbb{B}^*$, that maps a signal of the design and a time point from $T = \{0, \dots, d_{max}\}$ to a bitvector value. To reduce the number of potential antecedent subexpressions, a subset $IS \subseteq S \times T$ containing the *influencing signals* is calculated by a path analysis of the witness and the circuit. The initial property is then given by the formula

$$\Big(\bigwedge_{(s,t) \in IS} (s^t = a(s,t)) \Big) \rightarrow e, \tag{4}$$

where s^t denotes the value of signal s at time point t. This property holds trivially, since all signals are assigned to the value they have in the witness for e. Furthermore, all signals that influence the value of the signals in e, are included in IS. Each assignment to a signal in Formula (4) forms an antecedent subexpression. In order to obtain sufficient subsets of the assignments in a to satisfy the expected behavior e, the property analysis is applied to formula (4).

Example 2: Consider the circuit in Figure 2. It implements a simple logic unit, where the $ctrl$ input selects among the operations AND and OR of the values of input in in two successive cycles. The result is stored in register R_1 and it is shown at output out one cycle later. If rst is high, both registers are reset to zero. Using inverse property checking, we want to obtain a property for the target behavior e given by $\mathsf{next}[2](out) = in \vee \mathsf{next}(in)$, which describes the selected OR operation. The maximum delay d_{max} at output out is 2. In the first step a witness of length 3 is computed for e (see Figure 3). By a structural analysis of the circuit, the potential 18 assignments from the witness (6 signals in 3 cycles) are already reduced to the set of influencing signals

$$IS = \{(out, 2), (R_1, 2), (R_0, 1), (in, 0), \\ (in, 1), (rst, 0), (rst, 1), (ctrl, 1)\},$$

which is then used to construct the initial property according to Formula (4). After applying the property analysis on

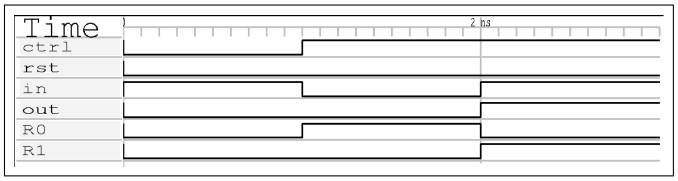


Fig. 3. Witness trace

```
property P = always(
    next_a[0..1](!rst) &&
    next(ctrl)
  ) -> (
    next[2](out) == (in | next(in))
  );
```

Fig. 4. Generated property for Example 2

the constructed initial property, further assignments could be discarded, resulting (among others) in the property shown in Figure 4. It states that when there is no reset within the first two cycles and $ctrl$ is high in the second cycle, then the computation will be performed as expected. This can be considered a correct operation and thus, a formal specification of the OR operation could be obtained automatically $\Box$

V. Integration with Coverage Analysis

For the above technology, the algorithm starts with a single witness for the expected behavior. The quality of the result depends on the assignment that is found by the underlying SAT solver. Thus, it is desirable to focus the search on interesting scenarios. As it turns out, the coverage analysis described in Section II-B provides exactly what is needed here.

With the approach from [5], it is checked for a design, if a set of properties covers the functionality of an output signal. If the coverage check fails, an uncovered scenario is presented in form of a counter-example which is basically an assignment to all the signals in the design. Now, it is common that the coverage check fails because of too strong assumptions or missing corner cases. Thus, it is also likely that one or more of the given properties matches the correct behavior for the uncovered scenario. This can be checked by simulating the involved properties with the assignment of the counter-example. If the consequent of a property holds for the scenario, it is indeed a witness and can be processed by the inverse property checking described above.

If none of the yet specified properties matches the scenario, a simple consequent expression can be derived from the concrete assignment to the target signal, that is currently checked for coverage. The generated properties will then show possible explanations of this assignment. In this way, the approach automatically presents valid modifications of the properties in order to include the yet uncovered behavior in the specification. By iteratively following this procedure full coverage is achieved much faster using the presented techniques.

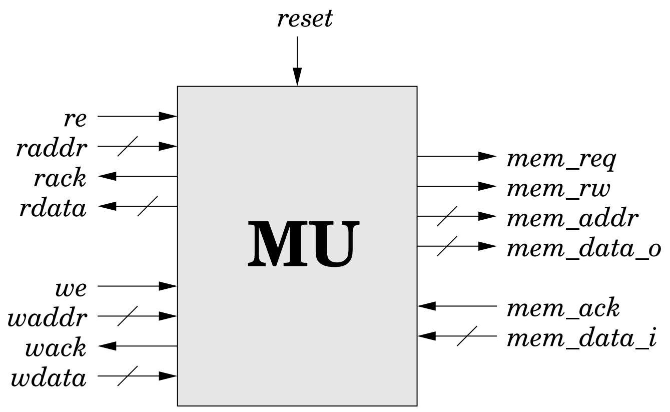


Fig. 5. Memory unit block view

TABLE I
MEMORY UNIT INTERFACE DESCRIPTION

	CPU side	
I	`reset`	reset
I	`re`	read enable
I	`raddr`	read address
O	`rack`	read acknowledge
O	`rdata`	data from mem
I	`we`	write enable
I	`waddr`	write address
I	`wdata`	data to mem
O	`wack`	write acknowledge
	Memory side	
O	`mem_req`	request
O	`mem_rw`	read or write
O	`mem_addr`	address
I	`mem_ack`	acknowledge
O	`mem_data_o`	data to mem
I	`mem_data_i`	data from mem
	Registers	
S	`state`	FSM state

VI. CASE STUDY

In this section it is demonstrated how the presented techniques can be used to assist the user in finding a formal specification of a hardware module. The design at hand is a memory unit (MU), that connects a CPU to a single port memory, thereby providing a dual port interface to the CPU (see Figure 5). It is implemented using a single write buffer, allowing for simultaneous read and write operations. The interface description of the MU is shown in Table I. The synthesized circuit consists of about $2,000$ gates.

The verification is started by examining the read operation of the MU. The initial simple read property is shown in Figure 6. It states that when the MU is in `IDLE` state, there is no reset, write enable is low and read enable is high, then a read access is performed on the memory side. The property holds for the implementation.

Now the *property analysis* is applied on property `READ`, revealing that in fact the expressions A_1, A_2 and A_4 are sufficient to prove the property. It can be concluded that even

```
property READ = always(
(A1) state == IDLE &&
(A2) !reset &&
(A3) !we &&
(A4) re
  ) -> (
    mem_req && !mem_rw && mem_addr == raddr
  );
```

Fig. 6. Initial read property for MU

```
property P0 = always(
     state == READ_WR_PEND &&
     !reset &&
     !mem_ack
  ) -> (
    mem_req && !mem_rw && mem_addr == raddr
  );
```

Fig. 7. Alternative read property

in case of a simultaneous read and write from the CPU, the read access will be performed on the memory side. This seems to be a reasonable behavior, as the write access can be stored in the internal write buffer at the same time. Note that this automatic strengthening of the properties can also point the verification engineer directly to a design bug that the original property would have missed, e.g. if the analysis states that a signal can be ignored that should definitely influence the behavior according to the specification.

Although the property READ could be strengthened automatically, it is certainly not a complete specification of the design. Nevertheless, the *coverage check* can be applied in order to obtain uncovered behavior that is either to be included in the specification or indicates a bug in the implementation. The coverage check for the signal `mem_req` returns an uncovered scenario that matches the proof goal of the property `READ`. The generated alternative property is shown in Figure 7. It can be interpreted as follows: during a read access with a pending write the read is kept active as long as the memory did not acknowledge it. So far this can be considered a correct behavior, and the property can be included in the property suite.

For further inspection of the design, *inverse property checking* is used to examine the write functionality. The expected behavior for a write access is presented as a PSL expression `(mem_req && mem_rw)`. The automatically generated properties are shown in Figure 8. Property P1 describes the behavior for a read access with pending write, where the read has been acknowledged by the memory. Thus, the MU can proceed to post the write access to the memory. P2 states that in absence of reset, whenever write enable is high and read enable is low, and after the last transaction has been acknowledged by the memory, the write access will be posted immediately, regardless of the internal FSM state. The last property P3 shows a slightly more complex scenario involving

```
property P1 = always(
    state == READ_WR_PEND &&
    mem_ack &&
    !reset
  ) -> (
    mem_req && mem_rw
  );

property P2 = always(
    mem_ack &&
    !reset &&
    !re && we
  ) -> (
    mem_req && mem_rw
  );

property P3 = always(
    state == IDLE &&
    next_a[0..1]( !reset ) &&
    re && we &&
    next( mem_ack )
  ) -> (
    next( mem_req && mem_rw )
  );
```

Fig. 8. Generated write properties

timing: starting from state IDLE with read and write enabled simultaneously, the write access will be processed in the next cycle, if the read access is acknowledged by the memory one cycle after it has been issued.

All properties hold by construction. Note that the knowledge on the design's behavior is obtained without the source code and without inspecting a wave trace. As for the examples above, the generated properties give a very concise description of the functionality, each involving only few signals. In this way, the signal mem_req could be covered after adding two more properties describing the buffering of a write access and the idle behavior of the MU.

Summarizing the above case study, the presented techniques can be used to give a better feedback to the verification engineer. The property analysis strengthens the written properties automatically. Using the coverage analysis, yet unspecified behavior is revealed that can then be integrated in the specification using inverse property checking.

VII. Conclusions

In this paper we have presented techniques to support the creation of a high quality property set in formal verification. Starting with simple properties with strong assumptions, these properties can be strengthened automatically using the property analysis. Furthermore, the analysis gives detailed information on which parts of the antecedent are sufficient to prove the property. Complementary to this technique, inverse property checking can be used to discover and understand uncovered behavior in the design under verification. Integrated with an existing coverage check, these techniques can reduce the number of iterations and the effort for design understanding on the way to obtain full coverage. As a case study, the techniques have been applied to a memory unit.

References

[1] U. Kühne, D. Große, and R. Drechsler, "Property analysis and design understanding," in *Design, Automation and Test in Europe*, 2009.

[2] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[3] K. Claessen, "A coverage analysis for safety property lists," in *Int'l Conf. on Formal Methods in CAD*, 2007, pp. 139–145.

[4] 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.

[5] 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, July 2008.

[6] M. Bartley, D. Galpin, and T. Blackmore, "A comparison of three verification techniques: directed testing, pseudo-random testing and property checking," in *Design Automation Conf.*, 2002, pp. 819–823.

[7] 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*, ser. LNCS, vol. 1579. Springer Verlag, 1999, pp. 193–207.

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

[9] M. 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, Nov. 2008.

[10] *Accellera Property Specification Language Reference Manual, version 1.1*, http://www.pslsugar.org, 2005.

[11] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. on Comp.*, vol. 35, no. 8, pp. 677–691, 1986.

[12] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints," *J. Autom. Reason.*, vol. 40, no. 1, pp. 1–33, 2008.