Property Analysis and Design Understanding

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*—Verification is a major issue in circuit and system design. Formal methods like bounded model checking (BMC) can guarantee a high quality of the verification. There are several techniques that can check if a set of formal properties forms a complete specification of a design. But, in contrast to simulation-based methods, like random testing, formal verification requires a detailed knowledge of the design implementation. Finding the correct set of properties is a tedious and time consuming process. In this paper, two techniques are presented that provide automatic support for writing properties in a quality-driven BMC flow. The first technique can be used to analyze properties in order to remove redundant assumptions and to separate different scenarios. The second technique – inverse property checking – automatically generates valid properties for a given expected behavior. The techniques are integrated with a coverage check for BMC. Using the presented techniques, the number of iterations to obtain full coverage can be reduced, saving time and effort.**

I. Introduction

In today's hardware design flows verification is a major issue concerning time and effort. With an increasing design size, automation and tool support of the 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 – here we focus on property checking – the functional behavior is described by properties which are checked on the design using symbolic techniques [1]. Although simulation-based verification is still widely used in industry, formal verification generally offers the highest quality.

Regarding the effort to set up a verification environment, it is the easiest to use random simulation. In this methodology, the *Design Under Verification* (DUV) is treated as a black box. For more sophisticated approaches like directed tests or constraint-based random simulation, some insight into the implementation is necessary. However, writing a high-quality property set that formally captures the specification requires a deep knowledge of the DUV internals. 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 [2].

There are several techniques which can be used to check that a set of properties covers the whole functionality of a design [3], [4], [5], thus ensuring that the written properties form a complete specification. If there is a verification gap, a scenario can be extracted that is not yet specified. This can be used to guide the verification towards complete coverage. But still the properties have to be refined and formulated manually.

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 this work, we propose techniques to aid the verification engineer in design understanding and to ease the formalization of the specification. The contributions of this work are as follows: First, we introduce a technique to automatically analyze a given property, identifying too strong constraints on the environment or the internal state of the DUV. It is a common way to start with relatively strict assumptions and refining the properties until the most general case is met. Using our 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. Second, based on this approach we present a technique to automatically generate 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.

If the proposed techniques are integrated with an existing 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. In Section III related work is discussed. The new techniques are explained in Sections IV and V. Finally, the main results of an experimental evaluation are given in Section VI.

II. Preliminaries

A. Verification Setting

Bounded Model Checking (BMC) has been introduced in [6]. In contrast to the original BMC, we use an *all states verification*, meaning that the properties are proven for arbitrary starting states [7].

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

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.

is given by: $\wedge_{i=0}^{c-1} T_\delta(s_i, s_{i+1}) \ \wedge \neg \ p$, 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 to 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 [8]). A property has the form of an implication $A \rightarrow C$. 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. Related Work

Different approaches for analyzing properties have been proposed. An important research direction in this context is *vacuity detection* (see e.g. [9], [10], [11], [12]). The basic idea is to check if a property has a subformula which is irrelevant to its satisfaction. In [13] the resulting information from such checks is used to tighten the properties, i.e. redundancy in the property suite is removed.

In the domain of SAT the problem has been investigated what causes the unsatisfiability of a set of clauses. Here approaches have been developed to compute an unsatisfiable core (or unsatisfiable subformula), i.e. an already unsatisfiable subset of the set of clauses [14], [15]. Recently these approaches have been extended to extract all minimal unsatisfiable subformulas (see e.g. [16]). How this method can be used for the proposed property analysis is detailed later in Section IV-B.

In [17], [18] temporal logic *query checking* has been proposed. Given a model and a temporal logic formula (the query) which contains a place-holder, then a solution to the query is a propositional formula that is satisfied in the model. This concept has been enhanced in [19] to *trigger querying* such that temporal scenarios are returned that satisfy the query. While the underlying question is quite similar to our inverse property checking, we combine our method with the property analysis and propose the integration in a formal coverage analysis flow. Hence, the proposed techniques are part of the comprehensive task of building a complete property suite.

In [20], [21], techniques for the extraction of properties from simulation traces were introduced. These methods rely on a large data base of simulation results, while our approach produces formal properties that hold by construction.

IV. Property Analysis

The verification of a hardware design is a complex and creative task. Starting with an RTL implementation and a specification, the verification engineer has to figure out how the abstract concepts of the specification can be mapped to the signals in the implementation. To get an understanding of the design behavior, she usually starts with simple properties including very restrictive assumptions on the configuration and environment. These properties are then refined manually until full coverage is achieved.

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 his properties as concise as possible, saving time for further iterations on the way to full coverage.

The idea of the presented 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 next section details the approach.

A. Algorithm

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 $A \rightarrow C$, 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

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

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:

$$Q_k = \bigwedge_{i=1}^{n} (d_i \vee A_i) \ \wedge \ (\sum d_i = k) \ \wedge \ \neg C \tag{2}$$

The design is unrolled within the involved time interval (omitted in equation 2). The found assignments to the d_i variables are stored in a BDD $\mathcal{E}$ (*Binary Decision Diagram* [22]) 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

Algorithm 1: propertyAnalysis

```
A' = reformulated antecedent
for (k = 1 ... n) do
    repeat
        find assignment for Q_k
        store cube of d_i in BDD E
        block assignment
    until UNSAT ;
compute BDD S = ¬E
compute set cover of S'
```

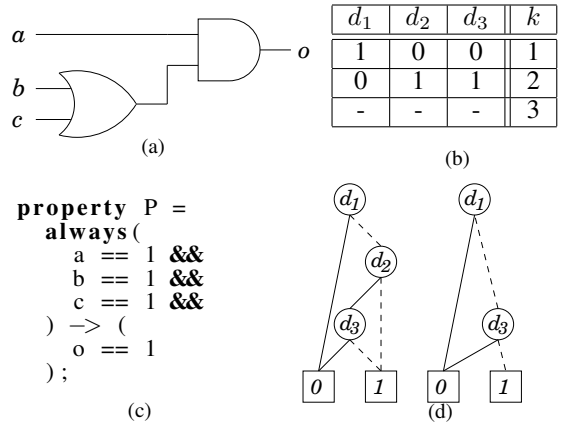

Fig. 1. Example for Algorithm 1

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

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 $A_1 = (a = 1)$ invalidates the property. The only solution for Q_2 is $(\neg d_1, d_2, d_3)$ and 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 $(a = 1)\&\&(b = 1)$.)

B. Discussion

There is a relation of the above algorithm to the calculation of *minimal unsatisfiable subformulas* (MUS) of a SAT instance. In [16], the authors point out the connection between *minimal correction sets* (MCS) of a SAT formula and MUS. A correction set is a set of clauses that turns an unsatisfiable formula into a satisfiable formula when being removed. All MUS form a hitting set of the MCS and vice versa. In our algorithm, the calculation of the essential subsets corresponds to the calculation of the MCS of the BMC instance with respect to the antecedent subexpressions. The extraction of sufficient antecedents from the BDD $\mathcal{S}$ is basically the construction of MUS from the complete MCS.

We have implemented the above analysis using an alternative algorithm based on the tool described in [16]. Obtaining equal quality of the results, we did not find that one approach outperforms the other one in terms of run-time. Using our approach, it is possible to perform an underapproximation on the obtained BDD $\mathcal{S}$ to filter out too long (and thus in most cases uninteresting) antecedents. It is subject to future investigations to figure out the best way of implementing the analysis.

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.

V. Inverse Property Checking

For a given property, the above analysis can be used to extract all the essential assumptions within the antecedent. But in case the user presents a property with assumptions describing a scenario which is far-off the design's core functionality, this does not guarantee that she captures the full behavior of the DUV. The analysis states which parts of the assumptions are essential for this specific scenario.

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 for that the given consequent expression holds. 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. In the following section the approach is described in more detail.

A. Algorithm

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, on the circuit the maximum delay d_{max} between any of the inputs or state signals and the signals involved in e is calculated. This is done 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 $T = \{0, \ldots, d_{max}\}$ 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

$$\Big(\bigwedge_{(s,t) \in IS} (s^t = a(s,t)) \Big) \rightarrow e, \tag{3}$$

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 3 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 3.

B. 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. The overall run-time of the enhanced verification flow is still dominated by the time for BMC. Regarding the techniques presented here, the most time is spent on the extraction of essential subsets of assumptions by successive calls to a SAT solver or using the technique from [16] (see Sections IV-A and IV-B, respectively).

VI. Experimental Evaluation

The presented techniques have been implemented and experimentally evaluated for different designs, among them a memory unit (MU). This MU connects a CPU to a single port memory, thereby providing a dual port interface to the CPU. Due to page limitation the details cannot be given here. Further information can be found in [23]. By applying the property analysis we were able to strengthen several of the written properties of the MU automatically. Moreover, using the coverage analysis in combination with the inverse property checking we were able to integrate yet unspecified behavior quickly. In summary, the number of iterations and the effort for design understanding on the way to obtain full coverage was reduced for the MU.

References

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

[2] 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.

[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] 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.

[7] M. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 11, pp. 2068–2082, Nov. 2008.

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

[9] I. Beer, S. Ben-David, U. Eisner, and Y. Rodeh, "Efficient detection of vacuity in ACTL formulas," in *Computer Aided Verification*, ser. LNCS, vol. 1254, 1997, pp. 279–290.

[10] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," in *Correct Hardware Design and Verification Methods*, 1999, pp. 82–96.

[11] M. Purandare and F. Somenzi, "Vacuum cleaning CTL formulae," in *Computer Aided Verification*, ser. LNCS, vol. 2404, 2002, pp. 485–499.

[12] A. G. J. Simmonds, J. Davies and M. Chechik, "Exploiting resolution proofs to speed up LTL vacuity detection for BMC," in *Int'l Conf. on Formal Methods in CAD*, 2007, pp. 3–12.

[13] H. Chockler and O. Strichman, "Easier and more informative vacuity checks," in *ACM & IEEE International Conference on Formal Methods and Models for Codesign*, 2007, pp. 189–198.

[14] E. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *Design, Automation and Test in Europe*, 2003, pp. 886–891.

[15] L. Zhang and S. Malik, "Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications," in *Design, Automation and Test in Europe*, 2003, pp. 880–885.

[16] 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.

[17] W. Chan, "Temporal-logic queries," in *Computer Aided Verification*, 2000, pp. 450–463.

[18] A. Gurfinkel, M. Chechik, and B. Devereux, "Temporal logic query checking: A tool for model exploration," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 898–914, 2003.

[19] O. Kupferman and Y. Lustig, "What triggers a behavior?" in *Int'l Conf. on Formal Methods in CAD*, 2007, pp. 146–153.

[20] G. Fey and R. Drechsler, "Design understanding by automatic property generation," in *Workshop on Synthesis And System Integration of Mixed Information technologies*, 2004, pp. 274–281.

[21] 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.

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

[23] U. Kühne, D. Große, and R. Drechsler, "Property analysis and design understanding in a quality-driven bounded model checking flow," in *IEEE International Workshop on Microprocessor Test and Verification*, 2008.