SAT-based Calculation of Source Code Coverage for BMC

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

Abstract

Property checking is the method of choice to guarantee functional correctness of a design under any input assignment and in any state. But so far only few methods to evaluate the coverage achieved by a set of properties have been presented. These methods either suffer from complexity problems known from CTL model checking or are incomplete themselves due to simulation-based engines.

In this work we present an approach to calculate coverage information in the context of *Bounded Model Checking* (BMC). The components of a design that are covered by a given set of properties are calculated. The result is presented at the source code level. The approach is explained in detail and empirically evaluated.

1 Introduction

Checking the functional correctness of a hardware design is a crucial issue in the design flow. Producing an erroneous design may cause a significant financial loss and damages the image of a design company. Traditionally, simulation-based methods were used to check the functional correctness of designs. But such methods can not cope with the huge state space of modern systems. Therefore methods for formal verification are applied. Formal property checking can ensure that a given property is valid under any input assignment and in any state of the design. But this is only valid for the properties considered. Therefore it has to be checked, if the given set of properties is sufficient to describe the design or which portions of the design are not covered [11]. In a practical application this coverage is usually estimated by manually reviewing all formal properties that have been applied.

Some work has been done to automate this process to improve the efficiency of the estimation and the reliability of the result. A rather informal approach to estimate coverage with respect to ’design intent’ has been proposed in [3]. This approach works by automatically combining multiple RT-level properties into higher-level properties. More formal measures have also been considered. A number of approaches consider CTL properties and calculate the states of the design that are covered [5, 9, 10]. But the computational complexity is the same as or even higher than that of CTL model checking. Moreover, these approaches have been proposed for designs given in form of finite state machines, but the relation to the original description of the design in a *Hardware Description Language* (HDL) is not considered. Therefore the practical application of such approaches is difficult.

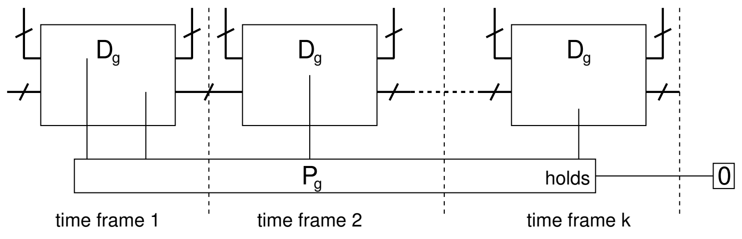

Figure 1: SAT instance for BMC

In [7] the coverage with respect to a high-level error model achieved by a given set of properties was considered. Bit-level errors are injected at the source code level. Then, fault simulation is applied for witnesses of the properties to calculate which properties fail under particular faults. When all errors of the fault model are detected by the properties, the property set is considered complete. Clearly, fault simulation for witnesses is less powerful than complete symbolic methods.

In this paper we propose an approach to automatically calculate the portions of a design that are covered by properties. Properties from *Bounded Model Checking* (BMC) [4] are considered. The coverage information is presented at the source code level. For this, the source code is split into components. A component is considered covered when changing the component allows to invalidate at least one property. A notion of coverage with a tighter definition is also presented and discussed. The calculation of covered components is based on techniques similar to model based diagnosis [15] or diagnosis approaches for the post-production test [17] and debugging [2]. All possible input assignments and states are considered when calculating coverage information. The computational complexity of the proposed approach is similar to BMC using *Boolean Satisfiability* (SAT). An example and benchmarks give empirical evidence of the feasibility of the approach.

2 Preliminaries

During BMC a property P and a design D are given. The property is usually described in some property specification language, e.g. Linear Time Logic [14] or Property Specification Language [1]. The design is described in a HDL, e.g. VHDL, Verilog, or SystemC. Then, a SAT instance is created that is satisfiable, iff the property fails. A satisfying assignment shows a trace that exhibits the failure, i.e. a counterexample to the validity of the property on the design.

In detail the SAT instance is built as shown in Figure 1. The property argues over time. Therefore the design D is mapped onto the gate-level representation D_g and unrolled for a bounded number of k time frames. Next state values calculated in one time frame are connected to the current state values of the next time frame. Thus, the observation of the sequential circuit is turned into a combinational problem. Then, the property is also transformed into a gate-level representation P_g and is connected to the appropriate signals in the different time frames of the unrolled circuit. The gate-level construct P_g has one output $holds$ that is one iff the property holds under the current input assignments and state assignments of the unrolled circuit. This output is constrained to value 0, i.e. the property does not hold. The whole problem is transformed into a SAT instance and a SAT solver is used to check if there exists a satisfying assignment. If no such assignment exists, the property holds under any assignment to inputs and states for the given bound k.

One way to proof the validity of a property regardless of a bound k is the use of induction as suggested in [16]. In the following such a technique is considered. But the coverage approach also applies to other types of BMC.

```
(1)  **Coverage**($D$, $\mathbb{P}$)
(2)  $\mathbb{C} = \emptyset$
(3)  Create gate-level representation $D_g$ of $D$ with annotated
     source code information
(4)  Associate covered predicates $c_1, \dots c_l$ to components in $D_g$
(5)  **for each** (property $P \in \mathbb{P}$)
(6)    Create a gate-level representation $P_g$ of $P$
(7)    Combine $D_g$ with covered predicates and $P_g$ into a
       SAT instance $S$
(8)    Add bounding constraint to $S$: $\sum_{i=1}^{l} c_i \leq 1$
(9)    **for each** (satisfying assignment $a$ of $S$)
(10)     **if** ($a(c_i) == 1$) $\mathbb{C} = \mathbb{C} \cup i$
(11) **return** $\mathbb{C}$
```

Figure 2: Algorithm to calculate coverage

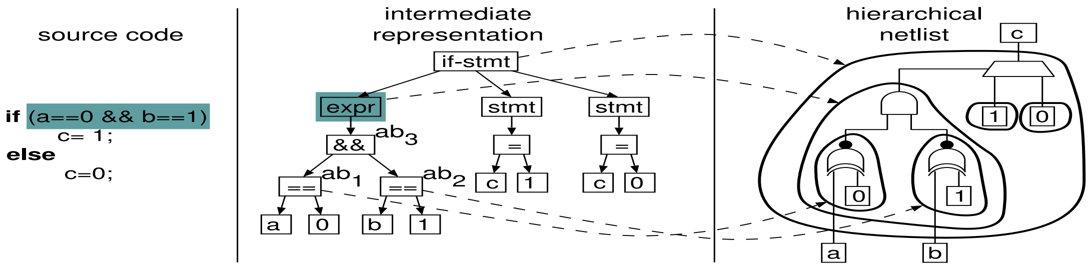

Figure 3: Synthesis and identification of components

3 Calculation of Coverage

The overall algorithm to calculate coverage at the HDL level is shown in Figure 2. The algorithm receives the design D and a set of properties $\mathbb{P}$ as input and returns the set $\mathbb{C}$ of components covered by the given properties. The number of components in the design is denoted by l. Initially, no component is considered as being covered (Line (2)). A gate-level representation of the circuit is created (Line (3)). The annotation of source code information is necessary to identify components in Line (4). This is explained in detail in Section 3.1. Then, the algorithm calculates for each property which components invalidate the property when being changed (Lines (5-10)). The necessary steps are considered in Section 3.2. In Section 3.3 a brief discussion follows.

3.1 Identification of Components

In order to calculate coverage information at the source code level a link between the HDL sources and the subsequently considered gate-level representation of the design has to be created. This is also used to identify components at the gate-level. The underlying technique was previously proposed in [6] and is briefly reviewed in the following.

The link between sources and gate-level is established during synthesis. This flow is shown in Figure 3. The syntactical structure of the source is given by an *Abstract Syntax Tree* (AST). During synthesis this AST is traversed and gates are created that implement particular expressions of the source code. Thus, parts of the source code correspond to nodes in the AST. The nodes, in turn, induce regions in the gate-level representation, where a region consists of a set of gates, of other regions, or both. Overall, the AST defines a hierarchical set of regions on the gate-level representation.

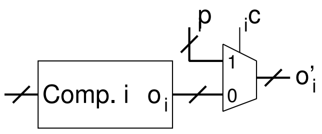

Figure 4: Covered attribute for component i

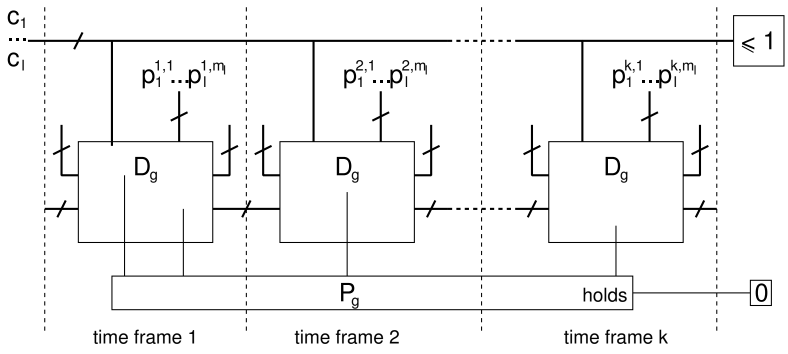

Figure 5: SAT instance for coverage calculation

For the coverage calculation these structures are also used to identify components. For example modules can be chosen as components, which leads to a coarse grain coverage information. A more fine grain coverage information is retrieved when all expressions are considered as components. Such syntactical constructs directly correspond to nodes of the AST and therefore identify sets of gates that are associated to one component.

In order to allow for a unique identification of gate-level regions with source-level constructs no optimization is applied during synthesis. For example when sharing an adder between then-branch and else-branch of an if-statement the unique identification would not be possible. Therefore in this context the term “synthesis” only refers to a direct mapping of source level constructs to the gate-level.

3.2 Coverage Information

A component is considered covered when changing the component invalidates at least one property of the set $\mathbb{P}$. A powerful fault-model is used to change a component: the function can be replaced by an arbitrary Boolean function. This can efficiently be calculated using a SAT solver.

As in model based diagnosis [15] or diagnosis for circuits based on SAT [17, 2] the function of a component is overridden, when a corresponding attribute is set. This is illustrated in Figure 4. The outputs o_i of component i are overridden by arbitrary pseudo inputs p_i if the covered attribute c_i is set to 1. Otherwise the values originally calculated by the component are propagated.

This construction is applied to each component induced by the AST in the gate-level representation of the design D_g when creating the SAT instance S (Figure 2, Line (6)). The other details are similar to BMC as introduced in Section 2. The original SAT instance was unsatisfiable. But now the resulting SAT instance is satisfiable. Changing the output of a component of D may cause the property to fail. By setting the corresponding attribute c_i to 1 and assigning appropriate values to p_i the SAT instance can be satisfied. A single component has only a single covered attribute that is used for all instances of the component and for all time frames in the unrolled design. In contrast different pseudo inputs are used for each instance and each time frame. Figure 5 shows the structure of the SAT instance S. The gate-level design D_g is unrolled for k time frames. Inputs, outputs and states of the circuit are not named in the figure to keep the presentation simple. The gate-level property P_g is denoted as a block below the unrolled design. Covered predicates are given as primary inputs $c_1, \ldots, c_l$ on the left hand side. The pseudo inputs for instance j of component i at time frame t are denoted by $p_i^{t,j}$.

Then, in Line (7) of Figure 2 a constraint is added to S to ensure that at most one component is changed. This additional constraint is shown on the right hand side in the structure of the SAT instance. For each satisfying assignment of S the covered components are added to the set $\mathbb{C}$ (Lines (8,9)).

3.3 Discussion

The complexity of the proposed approach is similar to that of SAT-based BMC. This is due to the construction. In the worst case two new variables are introduced per gate: the covered attribute and a new input value. For efficiency no multiplexer is included in the SAT instance S as indicated in Figure 4, but only the implication $\overline{c}_i \rightarrow o'_i = o_i$. The effect is the same, but the number of clauses and temporary variables is smaller.

All solutions SAT solvers have been proposed in [8, 12] that improve the enumeration of all solutions for a SAT instance. Instead of calculating completely specified solutions cubes that correspond to satisfying subspaces are enumerated. For this, satisfying assignments can be "reduced". But in the proposed approach blocking clauses consisting only of a single covered predicate are added to the SAT instance to remove solutions. With this knowledge no time consuming reduction of assignments is necessary.

As explained in Section 3.2 the proposed approach can be explained with respect to a fault model. Here, a fault is the replacement of a component's function by an arbitrary other function. A component is covered when a single fault with respect to this fault model invalidates at least one property. A large number of error models (e.g. gate-change errors, wire-change errors) can be embedded into the present error model. While the simulation-based approach in [7] is defined with respect to an arbitrary error model practical results are presented with respect to the bit coverage error model which can also be embedded into the more general fault model considered in this paper. Additionally, in [7] only a particular set of traces is considered to calculate coverage information while the proposed approach considers any possible trace for a given length k. Therefore the set of components $\mathbb{C}$ returned by the new approach proposed in this paper is a superset of the covered components calculated by [7].

Based on this observation the current approach can also be seen as calculating an "upper bound" for the set of covered components: a component c is covered by property P, if changing c in some way invalidates P. Thus, it is calculated if under all possible changes of c there is at least one change that causes an error. A similar problem formulation can be used to get a "lower bound" of the set of covered elements: a component c is *not* covered by P, if there exists some way to change c without invalidating P. Thus, it is calculated if under all possible changes there is at least one change that does not cause an error. As a result the set of components that are not covered by any property is returned. But the transformation of this formulation requires additional overhead to ensure that a component is really changed for a given satisfying assignment.

These observations show questions for future research. Next, experimental results are presented to evaluate the approach.

4 Experimental Results

In this section results for a counter are presented in detail as an example. Then, experimental data regarding run times and coverage information is reported. All experiments were run on an AMD Athlon 64 3500+ machine (2.2GHz, 1GB, Linux).

The proposed methodology has been implemented in C/C++. For synthesis and identification of components the Verilog frontend vl2mv that comes with VIS [18] has been modified. The SAT-based tool for property checking and calculating coverage information is based on induction similar to [16]. Zchaff [13] was used as the underlying SAT solver (Version of Nov. 2004).

```
(1)  module counter (clock, reset_i,
           start_i, modval_i, out_ro);
     . . .
(2)  always @ (posedge clock )
(3)  begin
(4)    if (reset_i==1_{RC}) begin
(5)      state_r = IDLE_{RI};
(6)      out_ro = 0_{RL};
(7)      high_r = 0_{R};
(8)      end
(9)    else begin
(10)     state_r = state_r_{IC};
(11)     case (state_r)
(12)       IDLE: begin
(13)         out_ro = 0_{LI};
(14)         if (start_i==1_{I}) begin
(15)           high_r = modval_i;
(16)           state_r = COUNT;
(17)           end
(18)         end
(19)       COUNT: begin
(20)         if (out_ro==high_r_{LC})
(21)           state_r = IDLE;
(22)         else out_ro= out_r+1_{LC};
(23)         end
(24)       endcase
(25)     end
(26)   end
(27) endmodule
```

Figure 6: Source code for the counter

pReset:
always(reset_i=1 $\rightarrow$ next(out_ro=0 && state_r=IDLE && high_r=0))

pLower:
always(out_ro<=high_r $\rightarrow$ next(out_ro$\leq$high_r))

pCount:
always((state_r=COUNT && out_ro<high_r && reset_i=0) $\rightarrow$ (next(out_ro)=out_r+1 && next(out_ro)$\leq$high_r))

pIdle:
always((state_r=IDLE && out_ro<high_r) $\rightarrow$ (next(state_r=IDLE) || (start_i=1 && next(out_ro=0 && state_r=COUNT)))

pIdle':
always((state_r=IDLE && out_ro<high_r) $\rightarrow$ **((reset_i==1 || start_i==0) &&** next(state_r=IDLE)) || (start_i=1 && next(out_ro=0 && state_r=COUNT)))

Figure 7: Properties for the counter

4.1 Counter

A simple counter is considered as an example. First, the counter is explained. Then, the coverage information resulting for some properties is discussed.

The Verilog source code of the counter is given in Figure 6. The coverage information annotated by underlining expressions is explained later. Input signals are marked by *_i*, registered signals by *_r* and the registered output by *_ro*. The counter has a synchronous reset signal. Upon receiving a start signal *start_i* the counter starts counting up to *modval_i*. While counting the module is in state *COUNT*, otherwise it is in state *IDLE*.

Four properties for this counter are shown in the PSL language [1] in Figure 7. For these properties the coverage information was calculated for expressions and assignments only. The results are marked in the source code. An expression covered by the property *pReset*, *pLower*, *pIdle* or *pCount* is underlined and the letter *R*, *L*, *I* or *C*, respectively, is annotated.

Consider the property *pLower*. The property proves that the output value is always smaller than the value given by *high_r*. One could assume, that this property also covers the transition into the state *IDLE* in Line (21). But this is not the case.

Moreover, while the properties cover the operations that are carried out in the two states, the transitions are not covered. This is obvious for the transition in Line (21), because this is not reflected by any property. It is more difficult to understand that the transition to *COUNT* in Line (16) is not covered. On first sight, the assignment seems to be covered by property *pIdle*. But changing Line (16) to *state_r= IDLE* would not invalidate the property – the property would be fulfilled due to the first part of the proof obligation *next(state_r=IDLE)*. In contrast the property

				PC	cov. modules				cov. expressions				all			
D	loc	P	k	time	time	#	#m.	%	time	#	#m.	%	time	#	#m.	%
counter	47	pReset	2	<1	<1	1	1	100	<1	12	4	33	<1	20	5	25
		pIdle	2	<1	<1	1	1	100	<1	12	3	25	<1	20	3	15
		pCount	2	<1	<1	1	1	100	<1	12	4	33	<1	20	10	50
		pLower	3	<1	<1	1	1	100	<1	12	4	33	<1	20	5	25
heap	150	pLT4	3	<1	<1	1	1	100	<1	64	6	9	<1	104	10	10
		pFull7	7	1	9	1	1	100	11	64	13	20	12	104	15	14
		pFull8	8	<1	14	1	1	100	13	64	13	20	17	104	15	14
		pFull9	9	2	23	1	1	100	18	64	13	20	20	104	15	14
		pEmpty	7	<1	8	1	1	100	9	64	7	11	10	104	8	8
		pNoError	50	1801	1576	1	1	100	3748	64	48	75	8117	104	79	76

Table 1: Experimental results for individual properties

pIdle' covers the assignment due to the more accurate description of the *IDLE* state. This allows the counter only to remain in state *IDLE* upon a reset or when not receiving a start-signal.

In summary the proposed technique automatically identifies parts of the source code that are not covered in the example.

4.2 Experimental Data

More experimental data is reported in Table 1. The columns D, loc, P, and k report the name of the design, the number of lines in the Verilog code, the property, and the number of time frames the circuit was unrolled, respectively. Then, the run time for property checking is reported. Next, coverage information for the module level (cov. modules), for expressions and assignments (cov. expressions), and finally for all components (all) is reported. In all cases the run time, the number of identified components (#), and the number of marked components (#m.) are given. All run times are measured in CPU seconds.

The run times are always negligible when the counter is considered. But it can be seen, that the number of covered components increases when a finer granularity is chosen. The same is true for the heap. The run time usually increases for finer granularities due to the growing search space when more covered attributes are introduced.

A similar behavior can be expected in general. The finer the granularity the more components are covered, but at the same time the percentage of covered components decreases.

5 Conclusions

In this work we presented an efficient method to calculate coverage information in the context of BMC. The method is complete in the sense that all input assignments and states are considered to calculate coverage information for a given property. The level of granularity for the coverage information can be chosen by the user. An example and experimental data show the practical relevance of the approach. The current formulation returns an upper bound of the set of covered components. A problem formulation to calculate a lower bound has also been presented.

In future work the relation between the formulations for calculating upper bound and lower bound will be analyzed in detail. Especially, the two approaches will be compared to approaches based on more restrictive error models as the one presented in [7]. Another issue is the adoption of problem specific heuristics to improve the calculation speed when larger benchmarks are considered.

References

[1] Accellera. *Property Specification Language – Reference Manual*. Accellera Organization Inc., 2004. Available at http://www.accellera.org/home.

[2] M.F. Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M.S.Abadir. Debugging sequential circuits using Boolean satisfiability. In *Int'l Conf. on CAD*, pages 204–209, 2004.

[3] P. Basu, S. Das, P. Dasgupta, P. Chakrabarti, C. Mohan, and L. Fix. Formal verification coverage: Are the RTL-properties covering the design's architectural intent? In *Design, Automation and Test in Europe*, pages 10668–10669, 2004.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.

[5] 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*, volume 2031 of *LNCS*, pages 528–543, 2001.

[6] G. Fey and R. Drechsler. Efficient hierarchical system debugging for property checking. In *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pages 41–46, 2005.

[7] F. Fummi, G. Pravadelli, and F. Toto. Coverage of formal properties based on a high-level fault model and functional ATPG. In *IEEE European Test Symposium*, pages 162–167, 2005.

[8] O. Grumberg, A. Schuster, and A. Yadgar. Memory efficient all-solutions SAT solver and its application to reachability. In *Int'l Conf. on Formal Methods in CAD*, volume 3312 of *LNCS*, pages 275–289, 2004.

[9] Y. Hoskote, T. Kam, P. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Design Automation Conf.*, pages 300–305, 1999.

[10] N. Jayakumar, M. Purandare, and F. Somenzi. Dos and don'ts of CTL state coverage estimation. In *Design Automation Conf.*, pages 292–295, 2003.

[11] S. Katz and O. Grumberg. Have I written enough properties - a method of comparison between specification and implementation. In *CHARME*, pages 280–297, 1999.

[12] B. Li, M.S. Hsiao, and S. Sheng. A novel SAT all-solutions solver for efficient preimage computation. In *Design, Automation and Test in Europe*, pages 10272–10278, 2004.

[13] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conf.*, pages 530–535, 2001.

[14] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[15] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[16] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 108–125, 2000.

[17] A. Smith, A. Veneris, and A. Viglas. Design diagnosis using Boolean satisfiability. In *ASP Design Automation Conf.*, pages 218–223, 2004.

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