Electronic Communications of the EASST
Volume 62 (2013)

Specification, Transformation, Navigation
Special Issue dedicated to Bernd Krieg-Brückner
on the Occasion of his 60th Birthday

Enhanced Formal Verification Flow for Circuits Integrating Debugging and Coverage Analysis

Daniel Große Görschwin Fey Rolf Drechsler

13 pages

Guest Editors: Till Mossakowski, Markus Roggenbach, Lutz Schröder
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Enhanced Formal Verification Flow for Circuits Integrating Debugging and Coverage Analysis

Daniel Große[1] **Görschwin Fey**[2] **Rolf Drechsler**[2]

[1] *solvertec GmbH*
Anne-Conway-Str. 1
28359 Bremen,Germany
grosse@solvertec.de

[2] *Institute of Computer Science*
University of Bremen
28359 Bremen, Germany
{fey,drechsle}@informatik.uni-bremen.de

Abstract: In this paper we briefly review techniques used in formal hardware verification. An advanced flow emerges from integrating two major methodological improvements: *debugging support* and *coverage analysis*. The verification engineer can locate the source of a failure with an automatic debugging support. Components are identified which explain the discrepancy between the property and the circuit behavior. This method is complemented by an approach to analyze functional coverage of the proven *Bounded Model Checking* (BMC) properties. The approach automatically determines whether the property set is complete or not. In the latter case coverage gaps are returned. Both techniques are integrated in an enhanced verification flow. A running example demonstrates the resulting advantages.

Keywords: Formal Verification, Boolean Satisfiability, Bounded Model Checking, Debugging, Functional Coverage

1 Introduction

For economic reasons the number of components in integrated circuits grows at an exponential pace according to Moore's law. This growth is expected to continue for another decade. Resulting is the so-called *design gap*, the productivity in circuit design does not increase as fast as the technical capabilities. Thus, more components can be integrated on a physical device than can be assembled in a meaningful way during circuit design. The *verification gap*, i.e. how to ensure the correctness of a design, is even wider. This is of particular importance in safety critical areas like traffic or security-related systems storing confidential information. Thus, techniques and tools for the verification of circuits received a lot of attention in the area of *Computer Aided Design* (CAD).

Simulation has always been in use as a fast method to validate the expected functionality of circuits. Once the circuit is described in a *Hardware Description Language* (HDL) like Verilog or VHDL, a testbench is used to excite the circuit under stimuli expected during in-field operation. But the full space of potential assignments and states of a circuit is exponential in the number of inputs and state elements. Therefore the search space cannot be exhausted by simulation. Even emulation using prototypical hardware is not sufficient to completely cover the full state space.

Thus, more powerful techniques for *formal verification* have been developed. In particular, to prove the correctness of a hardware design with respect to a given textual specification *property*

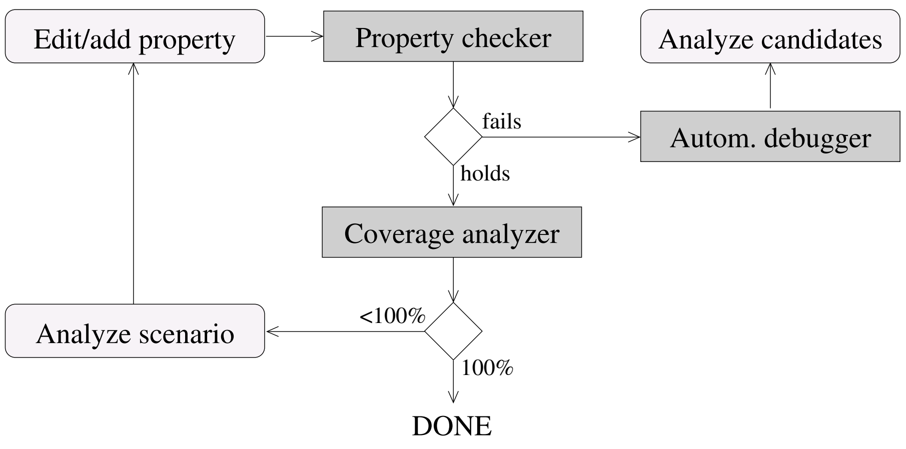

Figure 1: Enhanced flow

checking (or *model checking*) has been developed. Formal properties are derived from the textual specification. The properties are proven fully automatically to hold on the design. In *Symbolic Model Checking* [BCMD90, Bry95] *Binary Decision Diagrams* (BDDs) [Bry86] are used to represent the state space symbolically. This approach has been used successfully in practice. However, BDDs may suffer from memory explosion. As an alternative methods based on *Boolean Satisfiability* (SAT) have been proposed. In *Bounded Model Checking* (BMC) [BCCZ99] the system is unfolded for k time frames and together with the property converted into a SAT problem. If the corresponding SAT instance is satisfiable, a counter-example of length k has been found. Due to the significant improvements in the tools for SAT solving BMC is particularly effective. Even for very large designs meaningful properties can be handled [ADK$^+$05, NTW$^+$08]. Still the verification gap remains due to low productivity, and intensive training of verification engineers is required to apply property checking in practice. Therefore besides powerful reasoning engines, tool support is necessary for several tasks during formal verification.

Here we propose an enhanced verification flow enriched by techniques to ease the verification task. The flow is based on previous results by the authors. Figure 1 illustrates this flow. Dark boxes denote automatic tools while light boxes require manual interaction.

Typically only a property checker is available that returns a counter-example if a property fails. The subsequent debugging task is only supported by standard simulators. But techniques automating debugging in the context of property checking have been presented [FSBD08] to speed up the work flow. The debugger uses multiple counter-examples to determine candidate sites for the bug location and thus decreases the amount of manual interaction.

Another major problem in property checking is to decide when a property set is complete. This is usually done by manual inspection of all properties – a threat to correctness for any larger design. Thus, several methods to solve this problem have been proposed, e.g. [CKV03, Cla07, BBM$^+$07, Bor09]. An approach to automatically find uncovered scenarios has been presented in [GKD08]: The coverage analyzer determines whether the properties describe the behavior of the design under all possible input stimuli. If some input sequences are not covered, this scenario is returned to the verification engineer for manual inspection. Additional properties may be required or existing properties have to be modified.

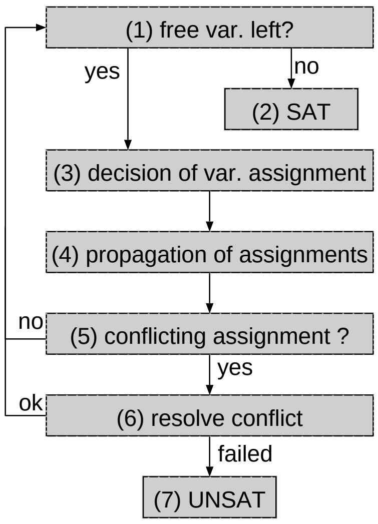
(1) free var. left?
yes
no
(2) SAT
(3) decision of var. assignment
(4) propagation of assignments
no
(5) conflicting assignment ?
yes
ok
(6) resolve conflict
failed
(7) UNSAT

Figure 2: DPLL algorithm in modern SAT solvers

This paper is structured as follows: The next section provides preliminaries on Boolean reasoning engines. Section 3 explains property checking as considered here. The automatic debugging approach is briefly discussed in Section 4. Section 5 describes the approach to automatically analyze functional coverage. An embedded example is used to illustrate the techniques. Section 6 concludes the paper.

2 Boolean Reasoning

Since the introduction of model checking there has been large interest in robust and scalable approaches for formal verification. Symbolic model checking based on Binary Decision Diagrams [McM93] has greatly improved scalability in comparison to explicit state enumeration techniques. However, these methods are impractical for industrial designs.

Due to dramatic advances of the algorithms for solving *Boolean Satisfiability* (SAT) many SAT-based verification approaches have emerged. Today, SAT is the workhorse for Boolean reasoning and is very successful in industrial practice [GG07]. Hence, in the following a brief introduction to SAT is provided as SAT is also the basis for *Bounded Model Checking* (BMC).

The SAT problem is defined as follows: Let h be a Boolean function in *Conjunctive Normal Form* (CNF), i.e. a product-of-sums representation. Then, the SAT problem is to decide whether an assignment for the variables of h exists such that h evaluates to 1 or to prove that no such assignment exists.

The CNF consists of a conjunction of clauses. A clause is a disjunction of literals and each literal is a propositional variable or its negation. SAT is one of the central NP-complete problems. In fact, it was the first known NP-complete problem that was proven by Cook in 1971 [Coo71]. Despite this proven complexity today there are SAT algorithms which solve many practical problem instances, i.e. a SAT instance can consist of hundreds of thousands of variables, millions of clauses, and tens of millions of literals.

For SAT solving several (backtracking) algorithms have been proposed [DP60, DLL62, MS99, MMZ+01, ES04]. The basic search procedure to find a satisfying assignment is shown in Fig. 2 and follows the structure of the DPLL algorithm [DP60, DLL62]. The name DPLL goes back to the initials of the surnames names of the authors of the original papers: Martin Davis, Hilary Putnam, George Logeman, and Donald Loveland.

Instead of simply traversing the complete space of assignments, intelligent decision heuristics [GN02], *conflict based learning* [MS99], and sophisticated engineering of the implication algorithm by *Boolean Constraint Propagation* (BCP) [MMZ+01] lead to an effective search procedure. The description in Fig. 2 follows the implementation of the procedure in modern SAT solvers. While there are free variables left (step 1), a decision is made (step 3) to assign a value to one of these variables. Then, implications are determined due to the last assignment by BCP (step 4). This may cause a conflict (step 5) that is analyzed. If the conflict can be resolved by undoing assignments from previous decisions, backtracking is done (step 6). Otherwise, the instance is unsatisfiable (step 7). If no further decision can be done, i.e. a value is assigned to all variables and this assignment did not cause a conflict, the CNF is satisfied (step 2).

To apply SAT for solving CAD problems an efficient translation of a circuit into CNF is necessary. The principle transformation in the context of Boolean expressions has been proposed by Tseitin [Tse68]. The Tseitin transformation can be done in linear time. This is achieved by introducing a new variable for each sub-expression and constraining that this new variable is equivalent to the sub-expression. For circuits the respective transformation has been presented in [Lar92].

Example 1 *Consider the expression* $(a+b)*c$ *where* $+$ *denotes a Boolean OR and* $*$ *denotes a Boolean AND. This is decomposed into two constraints:*

$$t_1 \leftrightarrow a + b \qquad (1)$$

$$t_2 \leftrightarrow t_1 * c \qquad (2)$$

These, are now transformed into CNF:

$$(t_1+\bar{a})*(t_1+\bar{b})*(\bar{t}_1+a+b) \qquad \text{(CNF for Constraint 1)}$$

$$(\bar{t}_2+t_1)*(\bar{t}_2+c)*(t_2+\bar{t}_1+\bar{c}) \qquad \text{(CNF for Constraint 2)}$$

Then, the final CNF for $(a+b)*c$ *is the conjunction of both CNFs where* t_2 *represents the result of the expression.*

3 Formal Hardware Verification using Bounded Model Checking

In the following the basic principles of BMC are provided. We use BMC [BCCZ99] in the form of interval property checking as described e.g. in [WTSF04, NTW+08]. Thus, a property is only defined over a finite time interval of the sequential synchronous circuit. In the following, the vector $s_t \in S$ denotes the states at time point t, the vector $i_t \in I$ the inputs and the vector $o_t \in O$ the outputs at time point t, respectively. The combinational logic of the circuit defines the next state function $\delta : I \times S \rightarrow S$ describing the transition from the current state s_t to the next state s_{t+1}

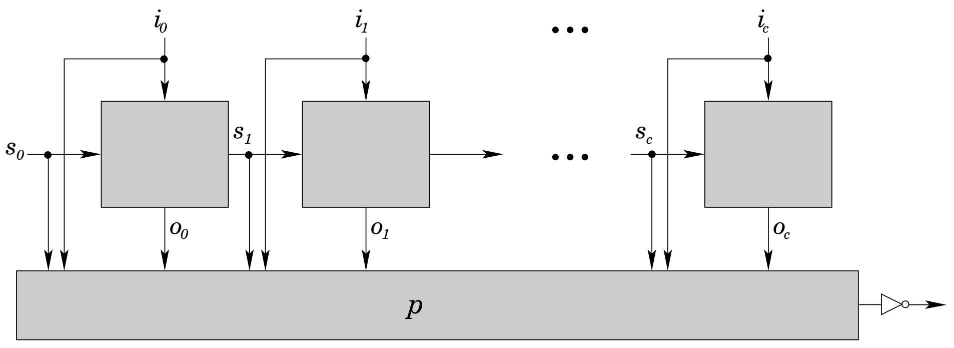

Figure 3: Unrolled circuit and property

under the input i_t. In the same way the output function $\lambda : I \times S \rightarrow O$ defines the outputs of the circuit. Then, a property over a finite time interval $[0,c]$ is a function $p : (I \times O \times S)^{c+1} \rightarrow \mathbb{B}$. For a sequence of inputs, outputs and states the value of $p(i_0, o_0, s_0, \ldots, i_c, o_c, s_c)$ determines whether the property holds or fails on the sequence. Based on the bounded property p the corresponding BMC instance $b : I^{c+1} \times S \rightarrow \mathbb{B}$ is formulated. Thereby, the state variables of the underlying *Finite State Machine* (FSM) are connected at the different time points, i.e. the current state variables are identified with the previous next state variables. This concept is called *unrolling*. In addition, the outputs over the time interval are determined by the output function of the FSM. Formally, the BMC instance for the property p over the finite interval $[0,c]$ is given by:

$$b(i_0, i_1, \ldots, i_c, s_0) = \bigwedge_{t=0}^{c-1} (s_{t+1} \equiv \delta(i_t, s_t)) \wedge \bigwedge_{t=0}^{c} (o_t \equiv \lambda(i_t, s_t)) \wedge \neg p$$

In Figure 3 the unrolled design and the property resulting in the defined BMC instance is depicted. As described in the previous section the BMC instance can be efficiently transformed into a SAT instance. As the property is negated in the formulation, a satisfying assignment corresponds to a case where the property fails – a counter-example.

In contrast to the original BMC as proposed in [BCCZ99] interval property checking does not restrict the state s_0 in the first time frame during the proof. This may lead to false negatives, i.e. counter-examples that start from an unreachable state. In such a case these states are excluded by adding additional assumptions to the property. 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.

In the following we assume that each property is an implication $always(A \rightarrow C)$. A is the antecedent and C is the consequent of the property and both consist of a timed expression. A timed expression is formulated on top of variables that are evaluated at different points in time within the time interval $[0,c]$ of the property. The operators in a timed expression are the typical HDL operators like logic, arithmetic and relational operators. The timing is expressed using the operators *next* and *prev*.

An example circuit given in Verilog and properties specified in the *Property Specification Language* (PSL) [Acc] are given in the following.

```
1  module sreg;
2  input clk, rst
3  input in, ctrl;
4  output out;
5  reg s0, s1;
6
7  assign out = s1;
8
9  always @(posedge clk)
10  if (rst) s0 <= 0;
11  else     s0 <= in;
12
13 always @(posedge clk)
14  if (rst) s1 <= 0;
15  else
16   //incorrect line:
17   //if (!ctrl)
18   if (ctrl)
19    s1 <= s0 || in;
20   else
21    s1 <= s0 && in;
22
23 endmodule
```

Figure 4: Example circuit

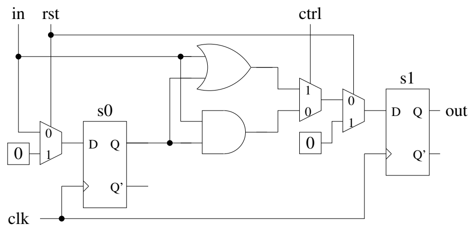

Figure 5: Logic representation of the circuit

Example 2 *Figure [4] describes the circuit* sreg *with the* reset *input and a* clock *signal, the data input* in *and the control input* ctrl*. The input data from input* in *is stored in the internal register* s0 *at each clock tick (lines 9–11). The control input* ctrl *decides whether output* out *returns the Boolean AND or the Boolean OR of the current value of* in *and the previously stored value* s0 *(lines 13-21). Figure [5] shows the logic level representation of this circuit.*

The unrolling of the circuit for two time steps is shown in Figure [6]. The flip flops are replaced by two signals to represent current state and next state. For example, the current state of $s0$ in time step 0 *is given by $s0_0$. Based on current state and primary input values the next value is calculated in $s0_1$.*

The simple PSL property pReset *in Figure [7] describes the behavior of this circuit immediately after a reset has been triggered. The property is defined over the interval $[0,1]$. The antecedent consists of a single assumption saying that the reset is triggered at time step 0 (line 2). The consequent specifies that the output is* 0 *in the next time step under this condition (line 4).*

Figure [8] shows a PSL property describing the standard operation of the circuit. The antecedent requires that no reset is triggered, i.e. rst == 0 *(line 2). Under this condition the output value after two time steps is defined by the value of the control signal in the next time step, the current value of the input and the next value of the input. If* ctrl *is zero (line 4), the output value after two time steps is the Boolean AND of the current input value and the next input value (line 5). Otherwise the output value is the Boolean OR of the two input values (line 6).*

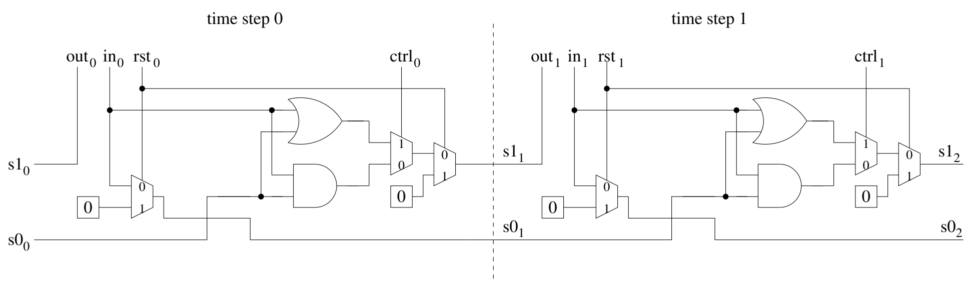

Figure 6: Circuit unrolled for two time steps

```
property pReset = always(
  rst == 1
) -> (
  next[1](out) == 0
);
```

Figure 7: Reset property

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

Figure 8: Property for normal operation

4 Debugging

As explained above, debugging is a manual task in the standard design flow. Tool automation helps to improve the productivity. An automatic approach for debugging in the context of equivalence checking has been proposed in [SVFA05] and extended to property checking in [FSBD08].

Essentially, the same model is created as for BMC shown in Figure 3. Additionally, for the failing property one or more counter-examples are given. The primary inputs of the circuit are restricted to this counter-example. While the property is restricted to hold. This forms a contradictory problem instance: when a counter-example is applied to the circuit, the property *does not* hold. Finding a cause for this contradiction yields a potential bug location, a so called fault candidate.

A fault candidate is a component in the circuit that can be replaced to fulfill the property. Here, a component may be a gate, a module in the hierarchical description, or an expression in the source code.

To determine such fault candidates, each component of the circuit is replaced by the logic shown in Figure 9. In the circuit component c implements the function f_c. This signal line is

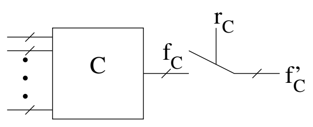

Figure 9: Repairing a component

modified to calculate $f_c' = !r_c \rightarrow f_c$, where r_c is a new primary input to the model. This allows to change the output value of c. When r_c is zero, c simply calculates the original output as given by the circuit description. When r_c is one, the circuit can be *repaired* by injecting a new output value for c.

A trivial solution at this stage would modify all components of the circuit at the same time and by this easily fulfill the attached property. Therefore an additional constraint limits the number of modifications. First a solution with only one modification is searched, if no such solution exists more modifications are iteratively allowed until a first solution has been found. For example, if more than one bug is present, often multiple modifications are required to fix the circuit. Then, all modifications that allow to fulfill the property are determined to retrieve all fault candidates. Finding the ”real” bug among the fault candidates is left to the verification engineer.

Some enhancements have been proposed to improve accuracy or efficiency of this simple approach [FSBD08]. Improving the accuracy can be achieved by using multiple counter-examples. The same construction as described above is done for all counter-examples. The same variables r_c are reused with respect to all counter-examples. Thus, the same components are modified to correct all counter-examples at the same time. Alternatively, the specification may be strengthened to improve the accuracy. By using multiple properties to specify correct behavior, the acceptable behavior is described more accurately. Therefore, false repairs are excluded. Finally, so called Ackermann constraints force all modifications to be realizable by combinational circuitry. The approach considered so far allows components to behave non-deterministic for repair, which is not feasible in practice. Ackermann constraints that force the same output assignment under the same input assignment for each instance of a component can be used to remove these infeasible fault candidates.

Efficiency can be improved, by incrementally using more and more counter-examples or more and more properties. Simulation-based preprocessing can help to remove some fault candidates in case of single faults.

Further works show how to improve the efficiency [SLM$^+$07, SFBD08], exploit hierarchical knowledge [FSV$^+$05], apply abstraction [SV07], fully correct a circuit with respect to a given specification [CMB07] or generate better counter-examples [SFB$^+$09].

Example 3 *Assume that the control signal* ctrl *of the Verilog circuit was interpreted wrongly. Instead of the correct line 18 in Figure 4, line 17 was used. In this case property* pOperation *in Figure 8*[1] *does not hold on the circuit. One counter-example may set* ctrl *to zero, so the* then*-branch of the* if*-statement is executed erroneously. The resulting output may be corrected by changing either line 19, where the operation is carried out, or line 17, the faulty condition. These two locations are returned as fault candidates by the approach. When adding another*

[1] Note that in the consequent of the property the conditional operator ? is used to express if-then-else.

counter-example that sets ctrl *to one, the* else-*branch is erroneously executed. Lines 17 or 21 are fault locations. Thus, only line 17 remains as a common fault candidate when both of the counter-examples are applied during automatic debugging.*

Experiments have shown that the number of fault candidates is reduced significantly compared to a simple cone-of-influence analysis [FSBD08].

5 Coverage Analysis

After debugging and finally proving a set of properties, the verification engineer wants to know if the property set describes the complete functional behavior of the circuit. Thus, in the standard design flow the properties are manually reviewed and the verification engineer checks whether properties have been specified for each output (and important internal signals) which prove the expected behavior in all possible scenarios. The coverage analysis approach introduced in [GKD08] automatically detects scenarios – assignments to inputs and states – where none of the properties specify the value of the considered output.

The general idea of the coverage approach is based on the generation of a *coverage property* for each output. If this coverage property holds, the value of the output o is specified by at least one property in any scenario. Essentially, the method shows that the union of all properties that involve the output o does not admit behavior else than the one defined by the circuit. For this task a multiplexor construct is inserted into the circuit for the actual output o, the properties describing the behavior of o are aligned and finally the coverage property for o is generated. Figure 10 depicts the final result of the multiplexor insertion which is carried out before unrolling. As can be seen for each bit of the n-bit output o a multiplexor is inserted. Each multiplexor is driven by the respective output bit and the inverted output bit. Then, in a renaming step the output of each multiplexor becomes o_i and the input o_orig_i, respectively. Moreover, the predicate sel computes the conjunction of all the select inputs of the multiplexers. Based on this construction, the coverage check can be performed by proving that the multiplexor is forced to select the original value of o (i.e. o_orig) at the maximum time point, assuming all properties involved. This is expressed in the generated coverage property of the considered output:

$$\left(\bigwedge_{j=1}^{|P_o|} \hat{p}_j \wedge \bigwedge_{t=0}^{t_{max}-1} X_t sel = 1 \right) \to X_{t_{max}} sel = 1,$$

where $|P_o|$ is the set of properties which involve o in the consequent, t_{max} is the maximum time point of time intervals defined by the properties in $|P_o|$, $\hat{p}_j$ are the adjusted properties over time, and X_k denotes the application of the next operator for k times. By this, the coverage problem is transformed into a BMC problem.

Complete coverage in terms of the approach is achieved by considering all outputs of a circuit. If all outputs are successfully proven to be covered by the properties, then the functional behavior of the circuit is fully specified.

Further works consider to aid the verification engineer while formulating properties with the goal to achieve full coverage. In [GWKD09] an approach to understand the reasons for contradictions in the antecedent of a property has been proposed. The technique of [KGD09] removes

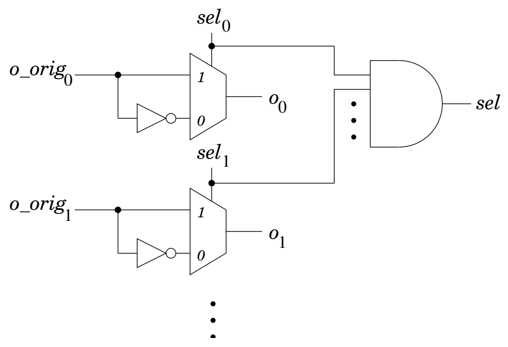

Figure 10: Insertion of the multiplexor

```
property pNew = always(
  (rst == 1) && next[1](rst == 0)
) -> (
  next[1](ctrl==0) ? next[2](out) == 0
    : next[2](out) == next[1](in)
);
```

Figure 11: Property required for full coverage

redundant assumptions in a property and generates valid properties for a given specified behavior. Both techniques can be combined with the coverage analysis to reduce the number of iterations to obtain full coverage.

Example 4 *Again consider the example circuit and the properties* pReset *and* pOperation. *Property* pReset *covers the behavior after the reset while property* pOperation *describes the normal operation of the circuit. Thus, full coverage should be reached and both properties are passed to the coverage analyzer.*

The algorithm unexpectedly returns the uncovered scenario where rst *is one in the first time step and zero in the next. Indeed none of the properties covers this case, because* pOperation *assumes* rst *to be zero in two consecutive time steps. Thus, a new property* pNew *to cover the remaining scenario is formulated as shown in Figure 11. Applying the coverage analysis to the three properties yields 100% coverage.*

In general, experimental evaluation has shown that the costs for coverage analysis are comparable to the verification costs [GKD08].

6 Conclusions

In this paper we have presented an enhanced formal verification flow. For formal verification the flow uses bounded model checking. The two major improvements in the new flow are the integration of debugging and coverage analysis. Both techniques automate manual tasks and hence the productivity improves significantly in comparison to a traditional flow.

Bibliography

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

[ADK+05] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, K. L. McMillan. An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment. In *CHARME*. Pp. 254–268. 2005.

[BBM+07] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, F. Bruno. Complete Formal Verification of Tricore2 and Other Processors. In *Design and Verification Conference (DVCon)*. 2007.

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*. LNCS 1579, pp. 193–207. Springer Verlag, 1999.

[BCMD90] J. Burch, E. Clarke, K. McMillan, D. Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conf.* Pp. 46–51. 1990.

[Bor09] J. Bormann. *Vollständige funktionale Verifikation*. PhD thesis, Technische Universität Kaiserslautern, 2009.

[Bry86] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comp.* 35(8):677–691, 1986.

[Bry95] R. Bryant. Binary Decision Diagrams and Beyond: Enabling Techniques for Formal Verification. In *Int'l Conf. on CAD*. Pp. 236–243. 1995.

[CKV03] H. Chockler, O. Kupferman, M. Y. Vardi. Coverage Metrics for Formal Verification. In *CHARME*. Pp. 111–125. 2003.

[Cla07] K. Claessen. A Coverage Analysis for Safety Property Lists. In *Int'l Conf. on Formal Methods in CAD*. Pp. 139–145. 2007.

[CMB07] K. Chang, I. Markov, V. Bertacco. Fixing Design Errors with Counterexamples and Resynthesis. In *ASP Design Automation Conf.* Pp. 944–949. 2007.

[Coo71] S. Cook. The complexity of theorem proving procedures. In *3. ACM Symposium on Theory of Computing*. Pp. 151–158. 1971.

[DLL62] M. Davis, G. Logeman, D. Loveland. A Machine Program for Theorem Proving. *Comm. of the ACM* 5:394–397, 1962.

[DP60] M. Davis, H. Putnam. A computing procedure for quantification theory. *Journal of the ACM* 7:506–521, 1960.

[ES04] N. Eén, N. Sörensson. An extensible SAT solver. In *SAT 2003*. LNCS 2919, pp. 502–518. 2004.

[FSBD08] G. Fey, S. Staber, R. Bloem, R. Drechsler. Automatic Fault Localization for Property Checking. *IEEE Trans. on CAD* 27(6):1138–1149, 2008.

[FSV$^+$05] M. Fahim Ali, S. Safarpour, A. Veneris, M. Abadir, R. Drechsler. Post-Verification Debugging of Hierarchical Designs. In *Int'l Conf. on CAD*. Pp. 871–876. 2005.

[GG07] M. Ganai, A. Gupta. *SAT-Based Scalable Formal Verification Solutions (Series on Integrated Circuits and Systems)*. Springer, 2007.

[GKD08] D. Große, U. Kühne, R. Drechsler. Analyzing Functional Coverage in Bounded Model Checking. *IEEE Trans. on CAD* 27(7):1305–1314, 2008.

[GN02] E. Goldberg, Y. Novikov. BerkMin: a Fast and Robust SAT-Solver. In *Design, Automation and Test in Europe*. Pp. 142–149. 2002.

[GWKD09] D. Große, R. Wille, U. Kühne, R. Drechsler. Contradictory Antecedent Debugging in Bounded Model Checking. In *Great Lakes Symp. VLSI*. Pp. 173–176. 2009.

[KGD09] U. Kühne, D. Große, R. Drechsler. Property Analysis and Design Understanding. In *Design, Automation and Test in Europe*. Pp. 1246–1249. 2009.

[Lar92] T. Larrabee. Test Pattern Generation Using Boolean Satisfiability. *IEEE Trans. on CAD* 11:4–15, 1992.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[MMZ$^+$01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conf.* Pp. 530–535. 2001.

[MS99] J. Marques-Silva, K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Comp.* 48(5):506–521, 1999.

[NTW$^+$08] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, W. Kunz. Unbounded Protocol Compliance Verification Using Interval Property Checking With Invariants. *IEEE Trans. on CAD* 27(11):2068–2082, 2008.

[SFB$^+$09] A. Sülflow, G. Fey, C. Braunstein, U. Kühne, R. Drechsler. Increasing the Accuracy of SAT-based Debugging. In *Design, Automation and Test in Europe*. Pp. 1326–1332. 2009.

[SFBD08] A. Sülflow, G. Fey, R. Bloem, R. Drechsler. Using Unsatisfiable Cores to Debug Multiple Design Errors. In *Great Lakes Symp. VLSI*. Pp. 77–82. 2008.

[SLM$^+$07] S. Safarpour, M. Liffton, H. Mangassarian, A. Veneris, K. A. Sakallah. Improved Design Debugging Using Maximum Satisfiability. In *Int'l Conf. on Formal Methods in CAD*. Pp. 13–19. 2007.

[SV07] S. Safarpour, A. Veneris. Abstraction and refinement techniques in automated design debugging. In *Design, Automation and Test in Europe*. Pp. 1182–1187. 2007.

[SVFA05] A. Smith, A. Veneris, M. Fahim Ali, A.Viglas. Fault Diagnosis and Logic Debugging Using Boolean Satisfiability. *IEEE Trans. on CAD* 24(10):1606–1621, 2005.

[Tse68] G. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part 2*. Pp. 115–125. 1968. (Reprinted in: J. Siekmann, G. Wrightson (Ed.), Automation of Reasoning, Vol. 2, Springer, Berlin, 1983, pp. 466-483.).

[WTSF04] K. Winkelmann, H.-J. Trylus, D. Stoffel, G. Fey. Cost-efficient Block Verification for a UMTS Up-link Chip-rate Coprocessor. In *Design, Automation and Test in Europe*. Volume 1, pp. 162–167. 2004.