Efficient Hierarchical System Debugging for Property Checking*

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

Abstract *Property checkers formally prove whether a given property holds for a circuit or system. In case of invalidity one or more counter-examples for the property are generated. Then the next step is debugging, a very time consuming task in complex designs that still lacks tool support. Often, the debugging phase is carried out using an ordinary simulator and requires a lot of manual work.*

In this paper we present an efficient automatic debugging approach for property checking to support the designer and verification engineer to easily identify the fault location. The technique fully supports hierarchical descriptions and by this enables diagnosis results at the source code level. Single and multiple errors are considered, while no assumption on a specific error type is needed. Debugging information is generated for the circuit and the property, as each of them can be erroneous.

1 Introduction

Due to the complexity of todays circuit and system designs verification becomes more and more important. Usually the design is given in a hardware description language, like Verilog or VHDL, and properties are described as assertions (e.g. in SystemVerilog) or in a dedicated property language as it is e.g. the case for PSL. Different techniques are available to check if the design obeys a given property. At the block-level typically formal techniques are applied. Here, property checking has become the state-of-the-art [19]. Simulation based techniques relying on assertions are usually used above the block-level. Additionally approaches that use formal techniques and simulation are available. When a failure of a property is detected all of these techniques usually present one counter-example, i.e. an execution trace in the sequential case, that shows this failure. As a next step debugging has to be carried out, where mostly simulators are used.

Recently, first approaches to improve debugging for property checking have been presented. Some of these methods focus on generation of counter-examples that are "more useful" for debugging [7, 13]. Others try to enhance the visualization and presentation of counter-examples [10, 5], or a combination of both [8]. But none of these approaches directly identify parts of the design as more or less likely to cause the failure.

This has been addressed earlier by error diagnosis in the fields of equivalence checking or circuit testing, e.g. [15, 3, 11, 18, 9, 12, 1]. But there approaches only consider the diagnosis problem on a flat netlist, i.e. on a combinational non-hierarchical gate-level description. Only a few approaches considered sequential [11, 6, 1] or hierarchical problems [3, 6], but these techniques do not consider the problems arising in the context of property checking.

*This work was supported in part by the *BMBF* project *VALSE-XT*.

Here for the first time a diagnosis based approach is presented to aid debugging the errors in a property or a design. The main features of this approach can be summarized as follows:

- The approach is simulation based and therefore very efficient. The core algorithm is based on an extension of *path-tracing*.
- Arbitrary types of errors can be handled as the approach is model free.
- Hierarchy is fully supported.
- Counter-examples over several time frames - and by this sequential problems - can be handled.
- The parts of the source code that are likely to cause the failure are identified.
- Debugging information is generated for the design and the property.

Several real world scenarios for errors at the source code level are considered and the advantages of the proposed approach are discussed. The property checking approach of [19] is used, but the debugging technique can easily be adapted to other property checking techniques.

2 Related Work

Even though our approach is the first one using diagnosis techniques to directly identify relevant parts of the source code, other ideas have been presented to aid debugging after property checking, to design diagnosis and to establish a source code link (see also Section 1 above). These techniques are discussed in more detail in this section, since they are often "orthogonal" to our method and can be used in combination or in the form of a preprocessing.

One approach to aid debugging for property checking was to improve the generation of counter-examples. Several works considered to minimize the number of value assignments in counter-examples, e.g. [16]. Here, the idea was to remove assignments from the counter-examples that are irrelevant for exciting the failure. Other approaches included the capability to produce counter-examples that are as similar as possible or as different as possible [7, 8]. In [13] a relation to the source code was established by partitioning signals into control-signals and non-control-signals. Based on this information an engine to calculate useful counter-examples was proposed. All these techniques can be used as a preprocessing to reduce the number of counter-examples needed for diagnosis. In this case our approach helps to interpret a given set of counter-examples afterwards. Our approach does not depend on time consuming symbolic operations, but is simulation based. The counter-examples can be generated by other techniques as well.

Furthermore tools for an enhanced presentation of counter-examples are available. In [10] a simulator with reasoning capabilities has been proposed to interactively analyze the cause of a value assignment or the outcome of a forced change of a signal value. An approach to enhance the presentation by the knowledge of several counter-examples and highlight differences and similarities between them was proposed in [5]. In [8] a mixed technique that applies both - improved counter-example generation and enhanced presentation - has been reported. These approaches are a complement to ours. They aid debugging, but none of them directly identifies parts of the source code as more or less important for debugging.

Approaches that directly identify parts of a circuit have been proposed for diagnosis in equivalence checking and testing of circuits. But only a few of them can be used for property checking, e.g. there are no structural similarities as suggested in [15] for the case of engineering change orders. Moreover for property checking a sequential problem has to be considered. Therefore BDD based approaches [9] might run into memory problems, when several time-frames are considered. In contrast simulation based [11, 3, 18, 12] and SAT based [1] diagnosis approaches can handle large designs and a large number of time-frames. A given set of counter-examples is the prerequisite to start diagnosis. Sequential problems have only been considered in [11, 6, 1]. But all these approaches only work on flat netlists.

In the approaches presented in [3, 6], regions were used to reflect hierarchy or proximity between gates. In our case a division of the circuit into regions enables the link of the diagnosis result to the source code and allows to attribute multiple gate-level errors to a single

location in the source code. None of the previous approaches from equivalence checking or test has been reported to establish such a link to the HDL-source. These approaches also do not handle the combination of circuit and property.

A method based on fault-diagnosis to establish a link from RTL to the gate-level has been proposed in [17]. But in that case the synthesis of the RTL-code was assumed to be disconnected from the diagnosis engine, which causes a decreased efficiency. In our application no disconnection between both tasks is necessary.

3 Preliminaries

In this section necessary notations and concepts are introduced to make the paper self-contained.

In the following a *circuit* (or *implementation*) C is considered to be a directed graph. Nodes of this graph represent logic gates. Edges of the graph represent the interconnection between gates. An order is imposed on the edges to determine the corresponding inputs of gates. The vectors of primary inputs, primary outputs, current states, next states and internal signals at time frame t are denoted by $I^t = (i_1^t, \ldots, i_n^t)$, $O^t = (o_1^t, \ldots, o_m^t)$, $S^t = (s_1^t, \ldots, s_l^t)$, $N^t = (n_1^t, \ldots, n_l^t)$ and $J^t = (j_1^t, \ldots, j_k^t)$, respectively. Where convenient these vectors are treated as sets instead.

A *property* P is a Boolean function p over variables $\bigcup_{t=0}^{\tau}(I^t \cup O^t \cup S^t \cup J^t)$, i.e. a Boolean function that relates the signals in different time frames of a corresponding circuit. The property can be viewed as circuit P with a single primary output p; primary inputs of P correspond to the variables. Figure 1 shows how the property and the circuit are combined into a circuit Γ for property checking. The circuit C is unrolled for τ time frames by connecting N^t to S^{t+1} for $t \in [0, \tau - 1]$. The inputs of P are attached to the corresponding signals in the unrolled circuit. If the output p is a tautology, the property is valid, otherwise the property is invalid. An assignment c to the primary inputs $\bigcup_{t=0}^{\tau}(I^t \cup S^0)$ of Γ that causes p to evaluate to 0 is a *counter-example* that shows the invalidity of property P in the circuit C.

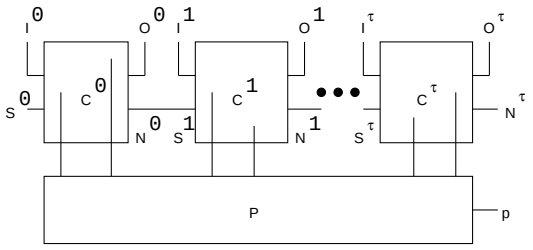

Figure 1: Combined circuit Γ

Remark 1 *When S^0 is restricted to be the set of initial states of C, bounded model checking as introduced in [2] is carried out. When S^0 is not restricted the type of property checking as applied in [19] results. The second type is considered in this work. However, the presented debugging technique applies to both types of property checking approaches as only counter-examples are considered.*

3.1 Path Tracing

The basic procedure for diagnosis is called *path tracing* and has first been used for diagnosis in the areas of equivalence checking [14] and testing [18]. Only the basic idea is outlined in this paper due to page limitation. The terms used in testing and equivalence checking are replaced by those known from property checking.

Given a single counter-example $c = (I, o)$ a set of candidate-errors sites is calculated using path tracing as explained in [18]. The procedure recursively marks all gates that would change the value of the erroneous primary output, when the value of the gate is changed. This procedure is iteratively carried out for all counter-examples in a given set Cex. A variable $m(g)$ is associated to each gate g to count the number of times the gate was marked by a counter-example. Thus, $m(g)$ indicates the number of counter-examples that could maximally be rectified by changing g.

This algorithm to calculate candidate error sites works on a flattened combinational circuit and has to find a few gate-level errors. In contrast to that, in property checking a sequential problem over a hierarchical circuit has to be handled and a link to the source code is necessary.

Debugging(circuit, property)

(1) Parse the source code and create an intermediate representation.
(2) Produce a hierarchical circuit from the intermediate representation.
(3) Instantiate submodules to create a flattened circuit.
(4) Unroll the flattened circuit.
(5) Connect property and circuit to create Γ.
(6) Carry out path tracing for every counter-example.
(7) Analyze the results of path tracing.

Figure 2: Overall debugging flow

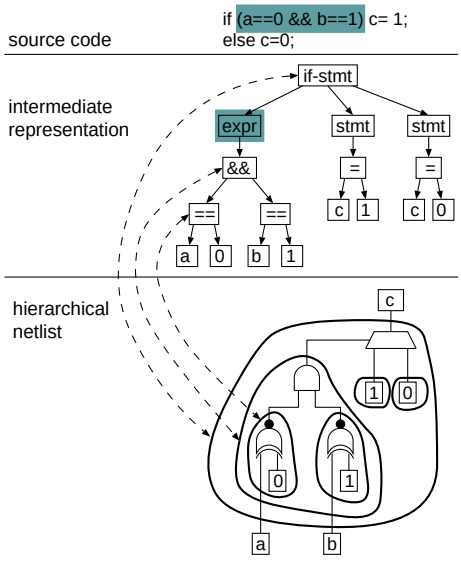

Figure 3: Link to the source code

Remark 2 *A single error in the source (e.g. a '$+$'-operation instead of a '$-$'-operation) usually translates to multiple gate-level errors: The set of gates implementing the subtraction should be replaced by another set of gates implementing an addition.*

4 Debugging Approach

This section explains how the property is handled and how sequentiality is addressed. Handling of hierarchy and linking multiple gate-level errors to the source code are discussed in the next section.

In equivalence checking or testing the question for debugging is

"*Why does output* o_i *assume the value* a *instead of value* a'*?*"

But in general the invalidity of a property can not be attributed to a single wrong output value of the circuit. Therefore the combination Γ of property and circuit as shown in Figure 1 has to be considered for debugging. Now, the question

"*Why does the property not hold?*"

can be translated into

"*Why does output* p *assume the value* 0 *instead of value* 1*?*"

Path tracing is used to calculate the set of gates that could change the value of p to 1 for a given a counter-example c.

Considering the combined circuit Γ has two advantages:

- The sequential dependencies of the problem are handled, because the implementation C is unrolled in Γ. Therefore also the influence of signals across time frames is detected by path tracing.
- Debugging information for the property is generated, because P is part of Γ. This eases the detection of a failure in the property.

5 Source Code Link

So far only the gate level has been considered. But usually the input for a property checker is given in a high-level description. The implementation C is described in Verilog or VHDL, while the property P is supplied in a property specification language or as an assertion in the source code. The property checker translates both into a Boolean representation and applies the checking algorithms at this level. Typically most information that relates the Boolean level to the sources is lost during this process. This section explains how the new approach handles source code and circuit as a unity and that this connection is necessary to relate multiple errors at the gate-level to a single error, e.g. a wrong operator, in the source code (see Remark 2).

The overall debugging flow proceeds as shown in Figure 2. In Steps (1) to (3) circuit and property are synthesized to create Γ. After these steps the origin of each gate in the

flattened circuit is available. Figure 3 illustrates this source code link. When the source code is parsed to build the intermediate representation, a part of the source code is associated to each node in the intermediate representation by storing positions from the source file for begin and end, e.g. the the node labeled *expr* corresponds to the if-condition as marked in the figure. Then, the intermediate representation is translated into a gate-level circuit. During this step the node that produced a gate is annotated at the gate. Therefore all gates that have been synthesized from the same node define a region. The bubbles in the figure indicate such regions. For some regions the corresponding node in the intermediate structure is connected by the dotted arrow. The tree structure of the intermediate representation is reflected in the superset and subset-relationships of the regions. The hierarchical circuit only contains each type of module once. A flattened circuit has to be created by instantiating all submodules (Step (3)). The flattened circuit is unrolled and the property is attached (Steps (4),(5)). Resulting is the circuit Γ. The connection to the source code is still available in Γ after Step (5). Then, path tracing is carried out for each counter-example (Step (6)). Let the directed acyclic graph $G = (V, E)$ represent the structure of the intermediate representation and let $V = \{v_0, \ldots v_\rho\}$ denote nodes in the intermediate representation. A single root node v_0 represents the whole circuit Γ. Then, each node v_i identifies a region R_i in the flattened circuit, i.e. a set of gates:

$$R_i := \{g : \text{Gate } g \text{ was synthesized from node } v_i\}$$

A counter $m(v_i)$ is associated to each node of the intermediate representation. When path-tracing is carried out $m(v_i)$ is increased if at least one node of R_i is marked by path tracing. The counter $m(v_i)$ is increased at most by one per counter-example. After iterating all counter-examples the value of $m(v_i)$ indicates how many counter-examples are at most rectifiable by changing the source code corresponding to v_i. The root node v_0 represents the whole circuit Γ, therefore $m(v_0)$ equals the number of counter-examples $|Cex|$ that have been used. Due to the subset relationships the counters of parents are always greater or equal compared to those of children, i.e. $m(v_i) \geq m(v_j)$ for $(v_i, v_j) \in E$.

After path-tracing the counters $m(v_i)$ have to be evaluated in Step (7) by traversing the intermediate representation. The traversal starts at the root node and visits only nodes with $m(v_i) > 0$. Marked are the parts of the source code that correspond to the bottom-most nodes that are visited (i.e. the nodes that correspond to the smallest regions and to the smallest parts of the source code). This approach is conservative; if a gate is marked by at least one counter-example, the corresponding source code is highlighted. By storing additional information it is possible to identify which part of the source code has been activated by which counter-example and in which time frame.

6 Error Scenarios

Several error scenarios are discussed in this section. The different scenarios are likely to occur during application of property checking. Due to page limitation no further experimental data to illustrate the efficiency of the approach can be given.

Data Path An error in the data path is due to a wrong operator or identifier in some expression. Therefore highlighting the erroneous expression is a useful hint for the designer in this case.

Control Path An error in the control path often results, when branching conditions are wrong. In this case the debugging tool highlights the conditions that have to be reviewed.

Hierarchy An error in a hierarchical description often leads to the following situation: The description of a submodule has a failure; each time the submodule is instantiated the failure is copied into the instance. The debugging tool should detect a single error location in the description of the submodule. Here, most other tools relying on single-error assumptions give very poor results.

Sequentiality Time dependent errors violate an expression of the type "*if condition α holds at time t_1, condition β holds at time t_2*". Essentially, this type of failure is an error in the data path or control path as well, but additionally the erroneous behavior only occurs, when several time frames are considered.

Erroneous Property In the previous scenarios the property was always assumed to be correct. But failing of a property is not always due to an error in the circuit. Instead the property itself can be erroneous. In fact, in practice the formulation of a property is not trivial. When a new complex property is described it is often incorrect. The decision, if the property or the circuit is incorrect *can not* be done automatically. The proposed approach handles the property like a part of the circuit. Therefore the same debug information is presented for the property.

7 Conclusions

In this paper the first approach to debugging for property checking has been proposed, that is able to cope efficiently with hierarchy, sequentiality and multiple errors, while making no assumption on the error type. Gate-level diagnosis techniques are enhanced to handle sequential and hierarchical designs. The diagnosis result is lifted from the gate-level to the source code level. By this multiple gate-level errors are recognized as a single error in the source code. Then, parts of the source that are likely to cause the error are highlighted. Different error scenarios have been discussed.

References

[1] M. 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.

[2] 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*. Springer Verlag, 1999.

[3] V. Boppana, R. Mukherjee, J. Jain, M. Fujita, and P. Bollineni. Multiple error diagnosis based on xlists. In *Design Automation Conf.*, pages 660–665, 1999.

[4] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[5] F. Copty, A. Irron, O. Weissberg, N. Kropp, and G. Kamhi. Efficient debugging in a formal verification environment. *Software Tools for Technology Transfer*, 4:335–348, 2003.

[6] A. D'Souza and M. Hsiao. Error diagnosis of sequential circuits using region based model. In *VLSI Design*, pages 103–108, 2001.

[7] G. Fey and R. Drechsler. Finding good counter-examples to aid design verification. In *MEMOCODE*, pages 51–52, 2003.

[8] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, number 3114 in LNCS, pages 453–456, July 2004.

[9] D. W. Hoffmann and T. Kropf. Efficient design error correction of digital circuits. In *Int'l Conf. on Comp. Design*, pages 465–472, 2000.

[10] Y.-C. Hsu, B. Tabbara, Y.-A. Chen, and F. Tsai. Advanced techniques for RTL debugging. In *Design Automation Conf.*, pages 362–367, 2003.

[11] S.-Y. Huang and K.-T. Cheng. Errortracer: Design error diagnosis based on fault simulation techniques. *IEEE Trans. on CAD*, 18(9):1341–1352, 1999.

[12] J.B.Liu, A.Veneris, and H.Takahashi. Incremental diagnosis of multiple open interconnects. In *Int'l Test Conf.*, 2002.

[13] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. *Software Tools for Technology Transfer*, 6(2):102–116, 2004.

[14] A. Kuehlmann, D. I. Cheng, A. Srinivasan, and D. P. LaPotin. Error diagnosis for transistor-level verification. In *Design Automation Conf.*, pages 218–224, 1994.

[15] C.-C. Lin, K.-C. Chen, S.-C. Chang, M. Marek-Sadowska, and K.-T. Cheng. Logic synthesis for engineering change. In *Design Automation Conf.*, pages 647–651, 1995.

[16] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 31–45. Springer Verlag, 2004.

[17] S. Ravi, I. Gosh, V. Boppana, and N. Jha. Faul-diagnosis-based technique for establishing RTL and gate-level correspondences. *IEEE Trans. on CAD*, 20(12):1414–1424, 2001.

[18] A. Veneris and I. N. Hajj. Design error diagnosis and correction via test vector simulation. *IEEE Trans. on CAD*, 18(12):1803–1816, 1999.

[19] K. Winkelmann, H.-J. Trylus, D. Stoffel, and G. Fey. A cost-efficient block verification for a UMTS up-link chip-rate coprocessor. In *Design, Automation and Test in Europe*, volume 1, pages 162–167, 2004.