Using BDD-based Decomposition for Automatic Error Correction of Combinatorial Circuits*

Dirk W. Hoffmann and Thomas Kropf

Institute for Computer Design and Fault Tolerance,
Prof. D. Schmid
University of Karlsruhe, D-76128 Karlsruhe, Germany
hoff@ira.uka.de kropf@ira.uka.de
http://goethe.ira.uka.de/hvg

Abstract. Boolean equivalence checking has turned out to be a powerful method for verifying combinatorial circuits and has been widely accepted both in academia and industry.
In this paper, we present a method for localizing and correcting errors in combinatorial circuits for which equivalence checking has failed. Our approach is general and does not assume any error model. Working directly on BDDs, the approach is well suited for integration into commonly used equivalence checkers.
Since circuits can be corrected fully automatically, our approach can save considerable debugging time and therefore will speed up the whole design cycle.
We have implemented a prototype verification tool and evaluated our method with the Berkeley benchmark circuits [7]. In addition, we have applied it successfully to a real life example taken from [11].
keywords: Automatic error correction, equivalence checking, BDDs, fault diagnosis

1 Introduction

In recent years, formal verification techniques [12] have become more and more sophisticated and for several application domains they have already found their way into industrial environments. Boolean equivalence checking [13, 4, 16], mostly based on BDDs [8, 9], is unquestionably one of these techniques and is usually applied during the optimization process to ensure that an optimized circuit still exhibits the same behavior as the original "golden" design. When using BDDs for representing boolean functions, the verification task mainly consists of creating a BDD for the boolean function of each output signal. Then, due to the normal form property of BDDs, both signals implement the same function if and only if they have the same BDD representation. Hence, equivalence can be decided by simply comparing both BDDs.

* This work is supported by the ESPRIT LTR Project 26241

A major requirement for successful application of formal methods in industrial environments is the ability of a verification tool to provide useful information even when the verification attempt fails. Thus the application domain of formal verification is no longer restricted to proving correctness of a specific design, but can also serve as a powerful debugging technique and therefore help speeding up the whole design cycle.

If equivalence checking fails, most verification tools only allow to compute a counterexample in the form of a combination of input values for which the output of the optimized circuit differs from its specification. Therefore, in many cases it remains extremely hard to detect the error causing components. Counterexamples as produced by most equivalence checkers can only serve as hints for debugging a circuit, while a deeper understanding of the design is still needed.

In recent years, several approaches have been presented for extending equivalence checkers with capabilities not only to compute counterexamples, but to locate and rectify errors in the provided designs. The applicability of such a method is strongly influenced by the following aspects:

- Which types of errors can be found ?
- Does the method scale to large circuits ?
- How many modifications in the original circuit are required to achieve a correct result ?
- Does the method perform well if both circuits are structurally different ?

Earlier research in the area of automatic error correction mostly focused on localizing single gate errors ("single error assumption"). Most of this work [21, 10, 18, 19, 22, 20] assumes a concrete error model based on a classification of typical design errors (e.g. [1]). Errors are divided into *gate errors* (*missing gate*, *extra gate*, *wrong logical connective*) and *line errors* (*missing line*, *extra line*). Each gate is basically checked against these error classes and only circuits with a single gate or line error can be rectified.

In [15] and [14], no error model is assumed. The method presented in [15] propagates meta-variables through the circuit. Erroneous single gates are determined by solving formulas in quantified propositional logic. However, the method is very time consuming and needs to invoke a propositional prover.

In [14], the implementation circuit and the specification circuit are searched for equivalent signal pairs and a back-substitution algorithm is used for rectifying the circuit. The success of this method highly depends on structural similarities between the implementation and the specification.

Incremental synthesis [3, 5] is a field closely related to automatic error correction. An *old implementation*, an *old specification*, and a *new specification* are given. The goal is to create a *new implementation* fulfilling the new specification while reusing as much of the old implementation as possible. In [5], structural similarities between the new specification and the old specification are exploited to figure out subparts in the old implementation that can be reused. The method is based on the structural analysis technique in [2] and the method presented in [4] which uses a test generation strategy to determine equivalent parts in two designs.

In this paper, we present a method for localizing and correcting errors in combinatorial circuits based on boolean decomposition. Basically, we try to determine the smallest component containing the erroneous parts in the optimized circuit. Once such a component has been localized, a circuit correction is computed and suggested to the designer. Circuit rectifications are computed in form of a BDD and then converted back to a net-list description. This is in contrast to techniques such as [14, 3, 5] which basically modify a given design by putting the implementation and specification together and "rewiring" erroneous parts.

Unlike [21, 10, 18, 19, 22, 20], our approach does not assume any error model. Thus, arbitrary design errors can be detected. Moreover, computed solutions are weighted by a cost function in order to find a minimal solution – a solution that requires minimal number of modifications in the implementation.

Our method directly works on BDDs which eases the integration into state-of-the-art equivalence checkers. Since we only make use of the abstract BDD representation of the specification circuit, the success of our algorithm does not depend on any structural similarity between the implementation and the specification. Therefore our technique can even be applied in scenarios where the specification is given as a boolean formula or directly in form of a BDD. This is in contrast to [14, 3, 5] where the result is highly influenced by the layout-structure of the specification circuit.

Our approach is orthogonal to other verification techniques such as structure comparison. Combining these techniques, our method can be applied to large designs.

This paper is organized as follows: In Section 2, we give a brief introduction to the theoretical background. Section 3 describes the boolean decomposition algorithm and Section 4 shows how this algorithm is applied to locate and correct erroneous parts of a circuit. Section 5 describes our prototype verification tool and discusses several benchmark examples. We close our paper in Section 6 with a summary and some remarks about further research.

2 Preliminaries

In the following, $f, g, h, \ldots$ denote propositional formulas and $X, Y, Z, \ldots$ represent propositional variables. We use the symbol $\equiv$ to denote logical equivalence between propositional formulas while $=$ is used for expressing syntactical similarity.

A *variable instantiation* σ maps every propositional variable to one of the truth values 0 or 1. σf is the formula obtained from f by replacing all variables by the truth value assigned by σ. Since σf does no longer contain any variable, either $\sigma f \equiv 0$ or $\sigma f \equiv 1$ holds.

The *positive* and *negative cofactor* of f, written as $f|_X$ and $f|_{\neg X}$, represent the functions obtained from f where X is substituted by the truth values 1 and 0, respectively. A formula f is said to be *independent* of X, if $f|_X \equiv f|_{\neg X}$.

$f_{\downarrow g}$ represents some boolean function that agrees with f for all valuations which satisfy g. For all other valuations, $f_{\downarrow g}$ is not defined and can be chosen freely.

Formulas can naturally be represented in form of syntax-trees or directed acyclic graphs if we allow to share sub-terms. Inner nodes are labeled with logical connectives while leafs are labeled with propositional variables. Let f and g be two formulas and v be a node in the syntax-graph of f. $f[v \leftarrow g]$ denotes the formula created from the syntax-graph of f where node v has been replaced by the syntax-graph of g. $[v \leftarrow g]$ is called a *term-substitution*.

BDDs [8,9] are a canonical representation of boolean functions. Formally, we define a BDD as a triple (V, E, l) where (V, E) is a rooted, directed acyclic graph with $|V| < \infty$. Every inner node has exactly two successor nodes called $then(v)$ and $else(v)$. The labeling function l labels every leaf node $v \in V$ with an element $l(v) \in \{0, 1\}$ and every inner node with a propositional variable. Every BDD depends on an ordering on the propositional variables occurring in it and must fulfill the ordering condition $l(v_1) < l(v_2)$ for every edge $(v_1, v_2) \in E$.

Every node of a BDD $\mathcal{B}$ containing n propositional variables recursively defines a corresponding boolean function $f_v^{\mathcal{B}} : \{0,1\}^n \rightarrow \{0,1\}$ by

$$f_v^{\mathcal{B}} = \begin{cases} l(v) & \text{if } v \text{ is a leaf} \\ \left[l(v) \wedge f_{then(v)}^{\mathcal{B}}\right] \vee \left[\neg l(v) \wedge f_{else(v)}^{\mathcal{B}}\right] & \text{otherwise} \end{cases}$$

We intuitively identify any BDD with the boolean function induced by its root node.

If isomorphic sub-trees are merged and nodes v with $then(v) = else(v)$ are eliminated, it can be shown [8] that for each fixed variable ordering, the resulting graph is a canonical representation for propositional formulas. In the rest of this paper, we implicitly assume that all BDDs are given in their canonical representations.

3 A BDD-based Decomposition-Algorithm

Assume we are given three propositional formulas f,g, and h. The pair (g, h) is called a decomposition of f, if there exists a variable X in g with

$$f \equiv g[X \leftarrow h] \tag{1}$$

If formulas f, g, and variable X are given, the decomposition problem is to compute a formula h satisfying (1).

Example 1. Consider $f = (A \wedge B) \vee A$ and $g = A \vee X$. $(g, A \wedge B)$ and (g, A) are both decompositions of f since $f \equiv g[X \leftarrow (A \wedge B)]$ and $f \equiv g[X \leftarrow A]$. Assuming $g = C \wedge X$, there exists no decomposition for f since there is no term h such that $f \equiv g[X \leftarrow h]$.

Example 1 shows that if a decomposition exists, there are usually more than one solution.

The question if there exists a decomposition can be decided according to the following lemma:

Lemma 1. *Let f and g be two propositional formulas. X is a variable occurring in g. Then, there exists a formula h with $f \equiv g[X \leftarrow h]$ if and only if*

$$f \wedge (g|_{\neg X} \leftrightarrow g|_X) \equiv g \wedge (g|_{\neg X} \leftrightarrow g|_X) \tag{2}$$

Proof. First, we proof the direction from left to right. We have to show that (2) holds for all variable instantiations σ. We distinguish two cases: If $\sigma(g|_{\neg X}) \not\equiv \sigma(g|_X)$, equation (2) is trivially true. If $\sigma(g|_{\neg X}) \equiv \sigma(g|_X)$, the value of σg is independent of X and therefore $\sigma g \equiv \sigma(g|_{\neg X}) \equiv \sigma(g|_X)$. Knowing $f \equiv g[X \leftarrow h]$, we can conclude $\sigma f \equiv \sigma(g[X \leftarrow h]) \equiv \sigma g$ and therefore (2) holds as well.

For the direction from right to left, we partially define h for all variable instantiations σ with $\sigma(g|_{\neg X}) \not\equiv \sigma(g|_X)$ as

$$\sigma h = \begin{cases} 0 & \text{if } \sigma(g|_{\neg X}) = \sigma f \\ 1 & \text{if } \sigma(g|_X) = \sigma f \end{cases} \tag{3}$$

Again, we distinguish two cases. If $\sigma(g|_{\neg X}) \equiv \sigma(g|_X)$, equation (2) reduces to $\sigma f \equiv \sigma g$ and since σg is independent of X, we know $\sigma f \equiv \sigma(g[X \leftarrow h])$. If $\sigma(g|_{\neg X}) \not\equiv \sigma(g|_X)$, we get $\sigma(g[X \leftarrow h]) \equiv \sigma(g[X \leftarrow \sigma h]) \equiv \sigma f$ by the definition of h.

Lemma 1 reflects the idea that we can find some h with $f \equiv g[X \leftarrow h]$ iff f and g agree on all valuations that are independent of X (expressed by $g|_{\neg X} \leftrightarrow g|_X$). For all other valuations, we can construct h according to equation (3).

Assuming that all propositional formulas are represented via BDDs, Lemma 1 shows how decomposability can be decided by simply applying basic BDD operations to f and g. Using BDDs, these operations have computation time which is polynomial in the number of BDD-nodes.

Now, we provide a simple algorithm computing a term h with $f = g[X \leftarrow h]$ if f and g are decomposable. The algorithm takes BDDs for f, g, and X and returns a BDD for h. Variable X always has to be the last variable in the current variable ordering. This assumption is crucial for the algorithm to compute correct results. Figure 1 shows the algorithm in more detail.

The following lemma states the correctness of the decomposition-algorithm.

Lemma 2. *Assume f and g are decomposable in respect to variable X. Then,* `decompose`(f, g, X) *computes a function h with $g[X \leftarrow h] \equiv f$.*

Proof. We proof the theorem by structural induction on the BDD of g. Base cases: 1. g is a leaf-node (either labeled with 0 or 1): Since f and g are decomposable, there exists some h' with $f \equiv g[X \leftarrow h']$. Because g is a leaf node, its truth-value does not depend on X. Hence, $f \equiv g[X \leftarrow h'] \equiv g[X \leftarrow h]$. 2. If g is a

function `decompose` $(f : \mathrm{BDD}, g : \mathrm{BDD}, X : \mathrm{VAR}) \longrightarrow (h : \mathrm{BDD})$
  begin
    if $g \equiv 0$ **return** 0
    if $g \equiv 1$ **return** 0
    if $g \equiv X$ **return** f
    if $g \equiv \neg X$ **return** $\neg f$

    $v := l(g)$ // *root-node-label of* g
    $h_1 :=$ `decompose`$(f|_v, g|_v)$
    $h_2 :=$ `decompose`$(f|_{\neg v}, g|_{\neg v})$
    $h := (v \wedge h_1) \vee (\neg v \wedge h_2)$
    return h
  end

Fig. 1. BDD-based decomposition-algorithm

node labeled with X, we know that $g \equiv X$ or $g \equiv \neg X$ since X is the last variable in the variable ordering. Thus, if $g \equiv X$, we get $g[X \leftarrow h] \equiv X[X \leftarrow f] \equiv f$. On the other hand, if $g \equiv \neg X$, we get $g[X \leftarrow h] \equiv (\neg X)[X \leftarrow \neg f] \equiv f$.
Induction step: Assume $f|_v \equiv g|_v[X \leftarrow h_1]$ and $f|_{\neg v} \equiv g|_{\neg v}[X \leftarrow h_2]$. Then,

$$\begin{aligned} g[X \leftarrow h] &\equiv ((v \wedge g|_v) \vee (\neg v \wedge g|_{\neg v}))[X \leftarrow h] \\ &\equiv (v \wedge g|_v[X \leftarrow (v \wedge h_1) \vee (\neg v \wedge h_2)]) \vee \\ &\quad (\neg v \wedge g|_{\neg v}[X \leftarrow (v \wedge h_1) \vee (\neg v \wedge h_2)]) \end{aligned}$$

Now, we perform case split on v.
$v = 0$: $g[X \leftarrow h] \equiv g|_{\neg v}[X \leftarrow h_2] \equiv f|_{\neg v} \equiv (v \wedge f|_v) \vee (\neg v \wedge f|_{\neg v}) \equiv f$.
$v = 1$: $g[X \leftarrow h] \equiv g|_v[X \leftarrow h_1] \equiv f|_v \equiv (v \wedge f|_v) \vee (\neg v \wedge f|_{\neg v}) \equiv f$.
In both cases, we get $g[X \leftarrow h] \equiv f$ which had to be proved.

Using cofactor computation, conjunction, and disjunction, the function returned by the decomposition-algorithm can be computed directly according to the following lemma. Although less intuitive, this lemma allows to compute the result much faster than algorithm 1, especially when dealing with large BDDs.

Lemma 3. *Assume f and g are decomposable in respect to variable X. Then,*

$$\texttt{decompose}(f, g, X) = (g|_X \wedge f \wedge \neg(g|_{\neg X})) \vee (\neg(g|_X) \wedge \neg f \wedge g|_{\neg X}) \qquad (4)$$

Proof. Base case: If $g \equiv 0$ or $g \equiv 1$, both sides of (4) are equivalent to 0.
If $g \equiv X$, we get $X|_X f \neg(X|_{\neg X}) \vee \neg(X|_X)\neg f X|_{\neg X} \equiv f \vee (0 \wedge f) \equiv f$.
If $g \equiv \neg X$, we get $\neg X|_X f \neg(\neg X|_{\neg X}) \vee \neg(\neg X|_X)\neg f \neg X|_{\neg X} \equiv (0 \wedge f) \vee \neg f \equiv \neg f$.
Induction step:

$$\begin{aligned} \texttt{decompose}(f, g, X) &= (v \wedge \texttt{decompose}(f|_v, g|_v)) \vee (\neg v \wedge \texttt{decompose}(f|_{\neg v}, g|_{\neg v})) \\ &\equiv v(g|_{vX} f|_v \neg(g|_{v\neg X}) \vee \neg(g|_{vX})\neg f|_v g|_{v\neg X}) \vee \\ &\quad \neg v(g|_{\neg vX} f_{\neg v} \neg(g|_{\neg v\neg X}) \vee \neg(g|_{\neg vX})\neg f_{\neg v} g|_{\neg v\neg X}) \end{aligned}$$

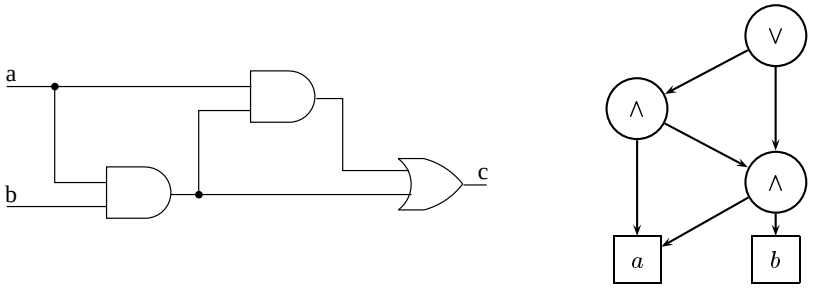

Fig. 2. The left picture shows some circuit-layout and the right picture shows the corresponding syntax-graph. Using directed acyclic graphs leads to a one-to-one correspondence between the circuit-layout and the formula-graph.

$$\begin{aligned}
&\equiv v(g|_X f\neg(g|_{\neg X}) \vee \neg(g|_X)\neg f g|_{\neg X})|_v \vee \\
&\quad \neg v(g|_X f\neg(g|_{\neg X}) \vee \neg(g|_X)\neg f g|_{\neg X})|_{\neg v} \\
&\equiv g|_X f\neg(g|_{\neg X}) \vee \neg(g|_X)\neg f g|_{\neg X}
\end{aligned}$$

4 Using Boolean Decomposition for Circuit-Rectification

In the following, assume we are given two circuits *spec* and *imp*. Let the propositional formulas f and g represent some output-signal of *spec* and *imp*, respectively, for which equivalence checking has failed (thus, $f \not\equiv g$). Further assume that formula g is directly extracted from its corresponding circuit description and represented in form of a directed acyclic graph. Using a syntax-graph instead of a syntax-tree leads to a one-to-one correspondence between the circuit-layout of *imp* and the syntactical representation of g (see Fig. 2). Whenever we talk of a circuit or net-list in the rest of this paper, we implicitely assume that it is represented in form of its corresponding syntax-graph.

Formula f is represented in form of a BDD only, since we exclusively make use of the abstract BDD representation of the specification circuit. Thus, computed results are totally independent of the layout-structure of *spec*. Since we do not consider the net-list of *spec* at all, our approach can even be applied in scenarios where the specification is given as a boolean formula or directly in form of a BDD.

Our goal is to modify the syntax-graph of g with a minimal number of changes such that $f \equiv g$ holds. Each such modification is called a rectification of g:

Definition 1. *Assume we are given two propositional formulas f and g with $f \not\equiv g$. ξ denotes some node in the syntax-graph of g. g is called rectifiable at ξ iff there exists a formula h with*

$$f \equiv g[\xi \leftarrow h] \tag{5}$$

The substitution of ξ by h is called a rectification of g.

The number of changes we have to apply to a given circuit is a crucial issue when computing rectifications since we want to preserve as much of the circuit structure as possible. In principle, we can always correct a wrong implementation by substituting the whole circuit by a DNF-representation of the specification-formula. Obviously, this is far away from what a designer would accept as circuit correction.

In practice, however, it is often possible to localize a comparably small sub-component containing all error-causing parts. This is obviously true, e.g., for all circuits fulfilling the single-error assumption. However, even if multiple errors occur in a given design, they are often concentrated in a single sub-component. Substituting this component can correct the circuit while preserving most of the circuit structure.

4.1 The Rectification Method

Our rectification-procedure is based on the boolean decomposition algorithms presented in Section 3 and mainly consists of two steps: the *location of erroneous sub-components* and the *computation of circuit corrections*.

For locating erroneous sub-components, we first try to figure out rectifiable sub-graphs in g. For doing this, we traverse the syntax-graph of g starting from the root. This directly corresponds to a back traversal of the circuit net-list starting from the corresponding output-signal.

For each node ξ, we determine if g can be rectified at ξ. According to Definition 1, we have to check if there is a formula h such that $g[\xi \leftarrow h]$ is logical equivalent to the specification formula f.

Replacing the sub-graph at ξ by a newly introduced variable X, we can easily perform this test by checking if there exists a term h such that $(g[\xi \leftarrow X], h)$ is a decomposition of f. For doing this, we first create a BDD-representation for f and $g[\xi \leftarrow X]$. Then, according to Lemma 1, decomposability can be decided easily by applying elementary BDD operations.

For computing circuit corrections, we first apply lemma 3 to compute a formula h such that $g[X \leftarrow h] \equiv f$. Since the algorithm only computes h in form of a BDD, it has to be converted back into a syntax-graph before the rectification can be applied to the circuit. This conversion, however, directly influences the resulting circuit structure and in order to minimize the number of modifications in g, we try to reuse as many sub-graphs of g as possible. Assume $g_1, \ldots, g_n$ are the sub-graphs of g we want to reuse. Hence, our goal is to create a syntax-graph for h containing $g_1, \ldots, g_n$.

To achieve this, we construct a second BDD h' as shown in Fig. 3. $G_1, \ldots, G_n$ are newly introduced BDD variables.

Since for all m,

$$\begin{aligned} h_{\downarrow g_1 \wedge \ldots g_{m-1}} &\equiv g_m h_{\downarrow g_1 \wedge \ldots g_{m-1} \wedge g_m} \vee \neg g_m h_{\downarrow g_1 \wedge \ldots g_{m-1} \wedge \neg g_m} \\ &\equiv (G_m h_{\downarrow g_1 \wedge \ldots g_{m-1} \wedge g_m} \vee \neg G_m h_{\downarrow g_1 \wedge \ldots g_{m-1} \wedge \neg g_m})[G_m \leftarrow g_m] \end{aligned}$$

the newly constructed BDD h' in Fig. 3 is logical equivalent to h if we substitute $G_1, \ldots, G_n$ by $g_1, \ldots, g_n$, respectively.

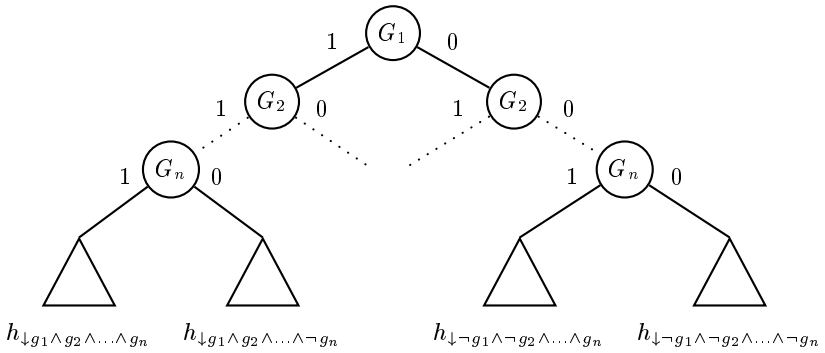

Fig. 3. Reuse of sub-graphs. Variables $G_1, \ldots, G_n$ denote newly introduced meta-variables representing sub-graphs $g_1, \ldots, g_n$, respectively.

A crucial issue in the construction process is to check the possibility if h can be exclusively constructed out of $g_1, \ldots, g_n$ and the logical connectives $\wedge$, $\vee$, and $\neg$. If h has this property, the sub-BDDs $h_{\downarrow f}$ in Fig. 3 can always be simplified to 0 or 1. Then, h' only contains the meta-variables $G_1, \ldots, G_n$. This property becomes important when dealing with hierarchical circuit descriptions. If we have located an erroneous sub-component in a circuit, we first try to replace it by another component that does not require changes in the component-interfaces. Thus, after computing a circuit correction h, we first try to convert h to a formula only involving the current component inputs as sub-terms. Every solution that keeps the component-interfaces unchanged is called *structure preserving*.

5 A Prototype Verification System

We have implemented a prototype equivalence checker integrating the methods and algorithms presented in this paper. Both the specification-circuit and the implementation-circuit have to be provided in an input language basically reflecting hierarchical net-list structures. A hierarchical description is achieved by defining various *components*. Each component consists of an *interface part* declaring input and output-variables as well as a *module body*. Besides additional component definitions for each sub-component, the module body contains a boolean formula for each output-variable defining its behavior in terms of input-variables and sub-component outputs.

As a toy-example, Fig. 4 shows the description of a two-bit carry-ripple adder. The circuit has four global input-signals a_1, a_2, b_1, b_2 and three output signals c_1, c_2, c_3. Using a half-adder (component `H_ADDER`) and a full-adder (component `FULL_ADDER`), the circuit computes the sum $(c_3 c_2 c_1) = (a_2 a_1) + (b_2 b_1)$.

The specification is shown in Fig. 5. Unlike the implementation, the specification defines its output-signals by boolean functions being derived directly from the truth-table of boolean addition.

```
COMPONENT CARRY_RIPPLE_ADDER (a1,a2,b1,b2) --> (c1,c2,c3)
        COMPONENT H_ADDER (a,b) --> (sum,carry)
                sum   := (a <-> b);
                carry := (a /\ b);
        END
        COMPONENT FULL_ADDER (a,b,c) --> (sum,carry)
                sum   := a XOR b XOR c;
                carry := (a /\ b) \/ (a /\ c) \/ (b /\ c);
        END
        H_ADDER.a := a1;
        H_ADDER.b := b1;
        FULL_ADDER.a := a2;
        FULL_ADDER.b := b2;
        FULL_ADDER.c := H_ADDER.carry;
        c1 := H_ADDER.sum;
        c2 := FULL_ADDER.sum;
        c3 := FULL_ADDER.carry;
END
```

Fig. 4. Example: A two bit Carry-Ripple-Adder

```
COMPONENT CARRY_RIPPLE_ADDER (a1,a2,b1,b2) --> (c1,c2,c3)
        c1 := (a1 XOR b1);
        c2 := (a2 XOR b2) XOR (a1 /\ b1);
        c3 := (a2 /\ b2) \/ (a1 /\ b2 /\ b1) \/ (a1 /\ a2 /\ b1);
        END
```

Fig. 5. Specification for the 2-bit adder-circuit.

After starting the equivalence checker, circuit-descriptions are parsed and converted into an internal representation. The user can then specify a pair of output-signals that are going to be compared. After calling the verification procedure, BDD representations for both output-signals are created. If the BDDs are different, the rectification algorithm as described in Section 4 is invoked and circuit corrections are computed.

Referring to our example, signals c_2 and c_3 can be proven to be equivalent on the first try. However, for signal c_1, equivalence checking fails and the verification tool tries to rectify the circuit automatically. Restricting ourselves to structure-preserving solutions, the verifier computes 6 different circuit corrections. For each solution, name of the sub-component to be modified and the number of required changes are displayed. The solution are weighted by a cost-function counting the number of modifications which have to be applied to the circuit. The solution that requires the minimal number of changes is displayed first. After the circuit has been rectified automatically, it can can be written back to a file.

Table 1 gives a summary of the computed solutions for the carry-ripple adder in Fig. 4. The second and third column contain name of the sub-component and

name of the signal to be modified, respectively. Column 4 reminds the old signal definition while column 5 shows the suggested replacement.

Nr	Component	Signal	old definition	suggested replacement
1	H_ADDER	sum	a ↔ b	a ↔ ¬b
2	H_ADDER	sum	a ↔ b	¬a ↔ b
3	CARRY_RIPPLE_ADDER	H_ADDER.b	b1	¬b1
4	CARRY_RIPPLE_ADDER	H_ADDER.a	a1	¬a1
5	H_ADDER	sum	a ↔ b	(a ∧ ¬b) ∨ (¬a ∧ b)
6	CARRY_RIPPLE_ADDER	c1	H_ADDER.sum	¬H_ADDER.sum

Table 1. Suggested circuit corrections for the carry-ripple adder.

Comparing Table 1 with the circuit-description in Fig. 4, it turns out that the major design error has been made in component H_ADDER. Output-signal sum computes a false value due to a wrong logical connective. Instead of performing an XOR-operation, the equivalence operator is applied. Solution 5 exactly suggests to replace this logical connective, but all other solutions also correct the circuit even if some of them actually do not reflect the designer's original intention. Since the verification tool does not have any semantical knowledge about the half-adder, it cannot distinguish between these solutions. In general, the solution that requires the minimal number of changes in the original circuit is considered best. Solutions 1 to 4 show that the circuit can even be rectified by inserting one additional NOT gate only.

A crucial aspect of the method is that only the abstract BDD of the specification is considered during the rectification-process. Therefore, the structure of the specification-circuit does not at all influence the computed solutions. Each specification-circuit – it's correctness assumed – causes the verifier to produce exactly the same results.

Table 2 shows measured data for the Berkeley benchmark circuits [7]. Arbitrary single gate errors have been introduced into the circuits and have been checked against the original designs. Column *rectification time* shows the elapsed time for analyzing and rectifying the circuit measured on a SUN Sparc Ultra 10 with 300 MHz and 128 MB main memory. Memory usage is shown in column *total BDD nodes*.

We also applied our method to a Galois-Field multiplier presented in [11] which we had to verify recently. When we first applied standard equivalence checking, verification failed and a deeper understanding of the multiplier had become necessary for correcting the circuit. Using our rectification-procedure, we have instantly been able to locate a missing NOT gate in the circuit layout shown in Fig. 3 in [11] whereas tedious manual search had been necessary before. The circuit could be rectified immediately (example *GFmult* in Table 2) and a lot of debugging time had been saved.

In general, runtime and memory usage of our method are mainly influenced by two factors: the number of *input signals* and the number of *internal gates*.

Since for some classes of formulas, BDDs grow exponentially in the number of input-signals, this value is the most limiting factor both for memory usage and runtime. The number of gates is also an important value since the rectification approach traverses the netlist and computes a BDD for each node in the graph. Thus, the number of gates considerably influences runtime of our approach especially when dealing with huge BDDs.

Our approach can be combined with other verification-techniques like structure-comparison and is therefore well suited for being integrated into state-of-the-art verification tools. A promising scenario is to eliminate similar parts of a circuit via structure-analysis and then to apply the rectification algorithm. In combination with these techniques, we believe that our method is applicable to industry-size examples.

name	inputs	gates	signal	rectification time	total BDD nodes
GFmult	6	8	c_out	< 0.01 sec	217
misj	35	57	91_out	0.01 sec	152
exep	29	386	71_out	0.01 sec	868
vg2	25	84	28_out	0.14 sec	9496
x1dn	27	108	32_out	0.14 sec	11294
x9dn	27	89	31_out	0.20 sec	11958
x6dn	38	285	42_out	0.89 sec	40791
jbp	36	397	87_out	0.58 sec	21409
chkn	29	511	539_out	0.83 sec	39608
signet	39	240	40_out	2.97 sec	224291
in6	33	188	41_out	0.12 sec	7604
in7	26	143	31_out	0.21 sec	13335
in3	34	302	49_out	0.43 sec	16501
in5	24	213	25_out	0.72 sec	37054
in	32	568	33_out	5.10 sec	162697
cps	24	936	942_out	4.86 sec	174852
bc0	21	952	927_out	8.21 sec	134424

Table 2. Experimental results. GFmult is taken from [11]. All other examples are taken from [7].

6 Summary

We have presented a method for localizing and correcting errors in combinatorial circuits for which equivalence checking has failed.

Unlike most other approaches, our method does not assume any error model. Thus, arbitrary design errors can be found. Our method is split into two parts: the *location of erroneous subcomponents* and the *computation of circuit corrections*. For both tasks, we have presented efficient solutions based on boolean decomposition. Working directly on BDDs eases the integration into commonly used equivalence checkers as a debugging back-end.

When computing circuit corrections, our approach tries to reuse as many parts of the old circuit as possible in order to minimize the number of modifications and therefore to increase the quality of the computed solutions.

We have implemented the presented methods in a prototype verification tool and evaluated it with the Berkeley benchmark circuits [7]. In addition, we have applied our method successfully to a real life example taken from [11]. Our method is powerful if the error causing elements are concentrated in a comparably small subpart of the circuit since our algorithm tries to locate the smallest subcomponent containing the erroneous components. This is obviously true, e.g., for all circuits fulfilling the single error assumption. Computed solutions are more expensive if the errors are widespread all over the circuit.

Our approach is orthogonal to other verification techniques such as structure comparison. Thus, the approach is well suited for being integrated into state-of-the-art verification tools. Combining these techniques, our approach can be applied to large designs.

In future, we plan to extend the verification system with front-ends for other input languages, such as BLIF [6] or PURR [17] to obtain access to a variety of example circuits. We further plan to incorporate our method in the PROSPER[1] project which aims at the integration of different proof tools into a higher-order logic environment. PROSPER tries to achieve a higher degree of automation and explicitly focuses on reducing the gap between formal verification and industrial aims and needs.

References

1. M.S. Abadir, J. Ferguson, and T.E. Kirkland. Logic design verification via test generation. *IEEE Transactions on CAD*, 7(1):138–148, January 1988.
2. D. Brand. The taming of synthesis. In *International Workshop on Logic Synthesis*, RTP, May 1991.
3. D. Brand. Incremental synthesis. In *Proceedings International Conference on COmputer Aided Design*, pages 126 – 129, 1992.
4. D. Brand. Verification of Large Synthesized Designs. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 534–537, Santa Clara, California, November 1993. ACM/IEEE, IEEE Computer Society Press.
5. D. Brand, A. Drumm, S. Kundu, and P. Narain. Incremental synthesis. In *Proceedings Internation Conference on Computer Aided Design*, pages 14–18, 1994.
6. R. K. Brayton, A. L. Sangiovanni-Vincentelli, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, S. Qadeer, R. K. Ranjan, T. R. Shiple, G. Swamy, T. Villa, G. D. Hachtel, F. Somenzi, A. Pardo, and S. Sarwary. VIS: A system for verification synthesis. In *Computer-Aided Verification*, New Brunswick, NJ, July-August 1996.
7. R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1986.
8. R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[1] http://www.dcs.gla.ac.uk/prosper/

9. R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
10. P.Y. Chung, Y.M. Wang, and I.N. Hajj. Diagnosis and correction of logic design errors in digital circuits. In *Proceedings of the 30th Design Automation Conference (DAC)*, 1993.
11. W. Drescher and G. Fettweis. VLSI Architectures for Multiplication in $GF(2^m)$ for Application Tailored Digital Signal Processors. In *Workshop on VLSI Signal Processing IX, San Francisco / CA*, 1996.
12. A. Gupta. Formal Hardware Verification Methods: A Survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.
13. Alan J. Hu. Formal hardware verification with BDDs: An introduction. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, pages 677–682, October 1997.
14. S.Y. Huang, K.C. Chen, and K.T. Cheng. Error correction based on verification techniques. In *Proceedings of the 33rd Design Automation Conference (DAC)*, 1996.
15. J.C. Madre, O. Coudert, and J.P. Billon. Automating the diagnosis and the rectification of design errors with PRIAM. In *Proceedings of ICCAD*, pages 30–33, 1989.
16. S.M. Reddy, W. Kunz, and D.K. Pradhan. Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment. In *ACM/IEEE Design Automation Conference*, pages 414–419, 1995.
17. K. Schneider and T. Kropf. The C@S system: Combining proof strategies for system verification. In T. Kropf, editor, *Formal Hardware Verification – Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*, pages 248–329. Springer Verlag, state of the art report edition, August 1997.
18. M. Tomita and H.H. Jiang. An algorithm for locating logic design errors. In *IEEE International Conference of Computer Aided Design (ICCAD)*, 1990.
19. M. Tomita, T. Yamamoto, F. Sumikawa, and K. Hirano. Rectification of multiple logic design errors in multiple output circuits. In *Proceedings of the 31st Design Automation Conference (DAC)*, 1994.
20. A. Wahba and D. Borrione. Design error diagnosis in sequential circuits. In Springer Verlag, editor, *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95*, volume 987 of *Lecture Notes in Computer Science*, Frankfurt(M), Germany, October 1995.
21. A. Wahba and D. Borrione. A method for automatic design error location and correction in combinational logic circuits. *Journal of Electronic Testing: Theory and Applications*, 8(2):113–127, April 1996.
22. A. Wahba and D. Borrione. Connection errors location and correction in combinational circuits. In *European Design and Test Conference ED&TC-97*, Paris, France, March 1997.