Automatic Error Correction of Large Circuits Using Boolean Decomposition and Abstraction[⋆]

Dirk W. Hoffmann and Thomas Kropf

Institut für Technische Informatik
Universität Tübingen, D-72076 Tübingen, Germany
{hoff,kropf}@informatik.uni-tuebingen.de

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. Thus, it allows the detection of arbitrary design errors. Since our method is *not* structure-based, the produced results are independent of any structural similarities between the implementation circuit and its specification. It can even be applied if the specification is given, e.g., as a propositional formula, a BDD, or in form of a truth table.
Furthermore, we discuss two kinds of circuit abstractions and prove compatibility with our rectification method. In combination with abstractions, we show that our method can be used to rectify large circuits.
We have implemented our approach in the **AC/3** equivalence checker and circuit rectifier and evaluated it with the Berkeley benchmark circuits [6] and the ISCAS85 benchmarks [7] to show its practical strength.

Keywords: Automatic error correction, design error diagnosis, equivalence checking, formal methods

1 Introduction

In recent years, formal verification techniques [11] 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.

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

[⋆] This work is supported by the ESPRIT LTR Project 26241 (PROSPER) the verification attempt fails. Then, the application domain of formal verification is no longer restricted to proving correctness of a specific design, but it can also be used as a debugging tool and therefore helps speeding up the whole design cycle.

If equivalence checking fails, most verification tools only allow the computation of counterexamples in the form of combinations of input values for which the output of the optimized circuit differ 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 needed for correction ?
- Does the method perform well if both circuits are structurally different ?

Most earlier research [20,10,17,18,21,19] in the area of automatic error correction assumes a concrete error model based on a classification of typical design errors going back to Abadir et. al. [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 most approaches can only handle circuits with exactly one error (*single error assumption*).

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 and abstraction. The main contributions of our approach can be summarized as follows:

- Unlike [20,10,17,18,21], our approach does not assume any error model. Thus, arbitrary design errors can be detected.

- Our method is *not* structure based. Only the abstract BDD representation of the specification is considered. Thus, the success of our algorithm does not depend on any structural similarity between the implementation and the specification. Our technique can even be applied in scenarios where the specification is given as a Boolean formula, a truth table, or directly in form of a BDD. This is in contrast to structure based methods such as [14,3,5] that can only be applied if both circuits are structurally similar.
- 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.
- 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 rectification procedure can be combined with circuit abstractions. In this paper, we discuss two kinds of circuit abstractions which are compatible with our method. We prove a correctness theorem showing how rectifications computed for abstract circuits can be lifted to circuit corrections for the original unabstracted circuits.
- We have implemented the rectification and abstraction algorithms in the **AC/3** [12] equivalence checker and circuit rectifier and evaluated the method with the Berkeley benchmark circuits [6] and the ISCAS85 benchmarks [7]. Our experimental results show that in combination with abstraction, our rectification 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 and Section 3 defines the formalism how combinatorial circuits are represented. Section 4 describes the rectification algorithm and Section 5 introduces circuit abstractions. Section 6 addresses the problem of rectifying multiple output circuits. We close our paper with experimental results in Section 7 and a conclusion in Section 8.

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.

The *positive* and *negative cofactor* of f, written as $f|_X$ and $f|_{\neg X}$, represent the functions obtained from f where X is instantiated 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.

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 $f \equiv g[X \leftarrow h]$.

3 Representing Combinatorial Circuits with Circuit Graphs

For the rest of this paper, we will use *circuit graphs* for representing combinatorial circuits. For a given circuit C, the corresponding circuit graph is constructed by introducing a new node v_c for each logical gate c in C. Wires are translated into transitions such that there is a transition from v_{c_1} to v_{c_2} iff some input of gate c_1 is connected with the output of gate c_2. Formally, we define circuit graphs as follows:

Definition 1. *A* circuit graph *$\mathcal{F}$ is a rooted, directed acyclic graph (V, l, e). V is a set of nodes with $|V| < \infty$. The function l labels every inner node of V with a logical connective and every leaf of V with a propositional variable. The* edge-function *e maps every node of $v \in V$ to an element $(v_1, \ldots, v_n) \in V^n$ where n is the arity of $l(v)$. The root-node of $\mathcal{F}$ is denoted by* root*($\mathcal{F}$).*

To simplify notation, we define the following abbreviations:

$$\begin{aligned} l(\mathcal{F}) &:= l(\mathit{root}(\mathcal{F})) \\ e_i(v) &:= i\text{-th element of } e(v),\ i \geq 1 \\ \mathcal{F}_i(v) &:= \text{sub-graph of } \mathcal{F} \text{ with root-node } e_i(v) \\ \mathcal{F}_i &:= \mathcal{F}_i(\mathit{root}(\mathcal{F})) \end{aligned}$$

Definition 1 is slightly stronger than the usual definition of labeled graphs where edges are represented by a relation $E \subset V \times V$. Using an edge-function e as defined above, multiple edges to the same successor node (Fig. 1 (a)) are possible. Furthermore, the successor nodes are implicitly ordered (Fig. 1 (b,c)). These properties cannot be expressed by using a relational edge-representation.

In the following, we restrict the set of logical connectives to $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (logical implication), $\leftrightarrow$ (logical equivalence), and $\oplus$ (exclusive-or). Every node v of a circuit-graph $\mathcal{F}$ induces a Boolean function f^v by

$$f^v = \begin{cases} l(v) & \text{if } v \text{ is a leaf} \\ \neg f^{e_1(v)} & \text{if } l(v) = \neg \\ f^{e_1(v)}\ l(v)\ f^{e_2(v)} & \text{if } l(v) \in \{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\} \end{cases}$$

To simplify notation we often identify a circuit graph $\mathcal{F}$ with its corresponding Boolean function and write $\mathcal{F}$ instead of $f^{\mathit{root}(\mathcal{F})}$ if it is clear from context.

We define two substitutions on circuit-graphs. *Node substitutions* replace a single node v in a graph $\mathcal{F}$ by some other graph $\mathcal{G}$ (denoted by $\mathcal{F}[v \leftarrow \mathcal{G}]$) while *variable*

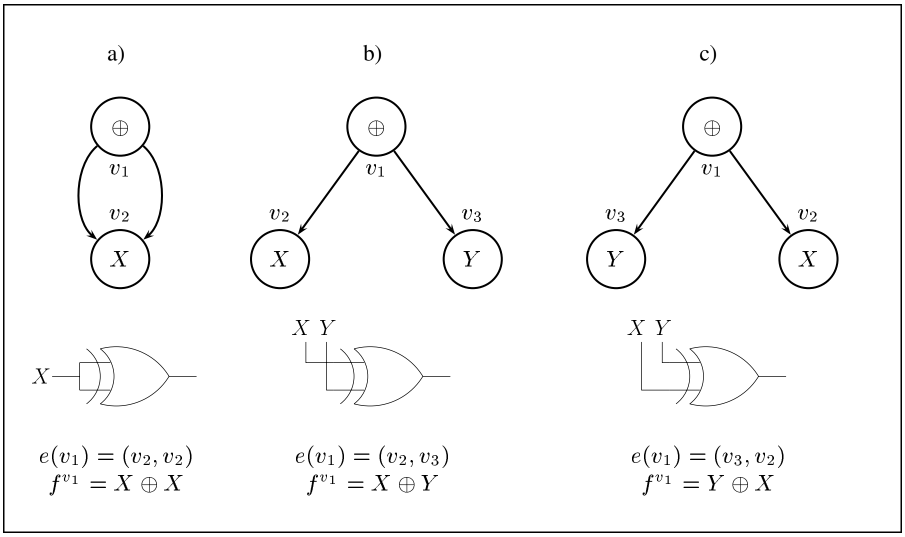


Fig. 1. Graph a) is a circuit-graph where an edge appears twice. Graph b) and c) demonstrate that an ordering on successors is necessary to distinguish the corresponding circuits.

substitutions replace every occurrence of some variable X by another graph $\mathcal{G}$ (denoted by $\mathcal{F}[X \leftarrow \mathcal{G}]$). If σ is the name of the substitution, we also write $\sigma\mathcal{F}$ instead of $\mathcal{F}[v \leftarrow \mathcal{G}]$ or $\mathcal{F}[X \leftarrow \mathcal{G}]$, respectively.

Note that node substitutions as defined here only replace a single node in the circuit-graph whereas variable substitutions replace *all* nodes that are labeled with the specified variable.

Lemma 1. *Let $\mathcal{F}, \mathcal{G}$ be two circuit-graphs. v is a node in $\mathcal{F}$. Then, for all variable substitutions σ,*

$$\sigma\left(\mathcal{F}[v \leftarrow \mathcal{G}]\right) = \left(\sigma\mathcal{F}\right)\left[v \leftarrow \sigma\mathcal{G}\right] \tag{2}$$

4 The Rectification Method

Let $\mathcal{G}$ be a combinatorial circuit represented as circuit graph. Further assume that $\mathcal{G}$ is a single output circuit. $\mathcal{F}$ denotes some Boolean function represented either as another combinatorial circuit (e.g., $\mathcal{G}$ could be the result of an optimization step applied to $\mathcal{F}$), a propositional formula, a truth table, or directly as a BDD. Since we only use the abstract BDD representation of $\mathcal{F}$ in our algorithm, the computed solutions are totally independent of the structure of $\mathcal{F}$. For the rest of this paper, we treat $\mathcal{F}$ as the specification and $\mathcal{G}$ as the implementation.

Our goal is to modify the circuit-graph of $\mathcal{G}$ with a minimal number of changes such that $\mathcal{F} \equiv \mathcal{G}$ holds. Each such modification is called a rectification of $\mathcal{G}$:

Definition 2. *Let $\mathcal{G}$ be a circuit-graph and $\mathcal{F}$ some Boolean function with $\mathcal{F} \not\equiv \mathcal{G}$. v denotes some node in the node-set of $\mathcal{G}$. $\mathcal{G}$ is called $\mathcal{F}$-*rectifiable at *v if there exists a circuit-graph $\mathcal{H}$ such that*

$$\mathcal{F} \equiv \mathcal{G}\,[v \leftarrow \mathcal{H}] \tag{3}$$

If $\mathcal{F}$ and v are clear from the context, we simply call $\mathcal{G}$ rectifiable.

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.

Our rectification procedure consists of two steps: the *location of rectifiable sub-graphs* and the *computation of circuit rectifications*. For locating rectifiable sub-graphs in $\mathcal{G}$, we traverse the circuit-graph of $\mathcal{G}$ starting from the outputs. In our implementation, we use a depth-first search strategy. For each node ξ, we determine if $\mathcal{G}$ can be rectified at ξ. According to Definition 2, we have to check if there is a formula $\mathcal{H}$ such that $\mathcal{G}[\xi \leftarrow \mathcal{H}]$ is logically equivalent to the specification $\mathcal{F}$. 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 $\mathcal{H}$ such that $(\mathcal{G}[\xi \leftarrow X], \mathcal{H})$ is a decomposition of $\mathcal{F}$. For doing this, we first create a BDD-representation for $\mathcal{F}$ and $\mathcal{G}[\xi \leftarrow X]$. Then, decomposability can be decided with standard BDD operations according to the following lemma which is a direct result from the theory of Boolean equations:

Lemma 2. *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*

$$f \wedge (g|_{\neg X} \leftrightarrow g|_X) \equiv g \wedge (g|_{\neg X} \leftrightarrow g|_X) \tag{4}$$

Basically, Lemma 2 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 $g|_{\neg X} \leftrightarrow g|_X$).

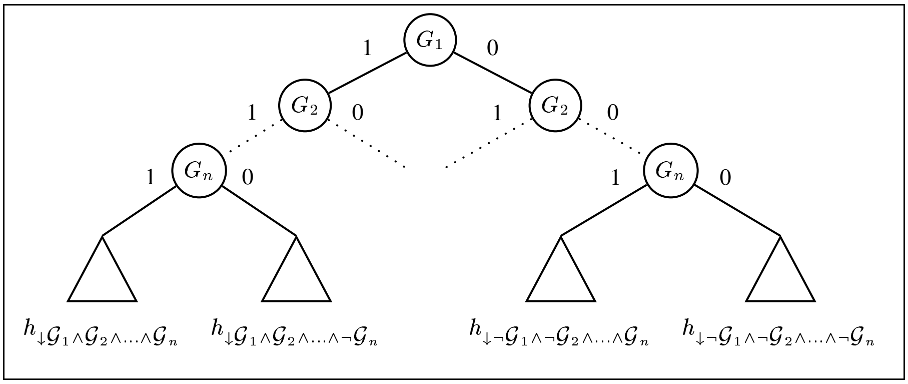
For computing circuit corrections, we first compute a formula $\mathcal{H}$ such that $\mathcal{G}[\xi \leftarrow \mathcal{H}] \equiv \mathcal{F}$. Again, this can be done by applying elementary BDD operations as the following lemma states:

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

$$f \equiv g\,[X \leftarrow [(g|_X \leftrightarrow f) \wedge (g|_X \oplus g|_{\neg X})]] \tag{5}$$

In Lemma (5), the solution formula is obviously not unique. Here, the solution with the smallest 1-set is being computed.

Using BDDs for representing Boolean functions, the application of Lemma 3 returns a formula h which is also represented in form of a BDD. Thus, the BDD for h first has to be converted back into a circuit-graph $\mathcal{H}$ before the rectification can be performed. This conversion, however, directly influences the resulting graph structure. To maximize the syntactical similarities between $\mathcal{G}$ and $\mathcal{G}[\xi \leftarrow \mathcal{H}]$, we try to reuse as many sub-graphs of $\mathcal{G}$ as possible. The heuristic implemented in **AC/3** is to reuse the current gate inputs or the set of inputs of the component containing ξ (when dealing with hierarchical circuits). Assume $\mathcal{G}_1, \ldots, \mathcal{G}_n$ are the sub-graphs of $\mathcal{G}$ we want to reuse. Hence,



G_1
G_2
G_2
G_n
G_n
1
0
$h_{\downarrow \mathcal{G}_1 \wedge \mathcal{G}_2 \wedge \ldots \wedge \mathcal{G}_n}$
$h_{\downarrow \mathcal{G}_1 \wedge \mathcal{G}_2 \wedge \ldots \wedge \neg \mathcal{G}_n}$
$h_{\downarrow \neg \mathcal{G}_1 \wedge \neg \mathcal{G}_2 \wedge \ldots \wedge \mathcal{G}_n}$
$h_{\downarrow \neg \mathcal{G}_1 \wedge \neg \mathcal{G}_2 \wedge \ldots \wedge \neg \mathcal{G}_n}$

Fig. 2. Reuse of sub-graphs. Variables $G_1, \ldots, G_n$ denote newly introduced meta-variables representing sub-graphs $\mathcal{G}_1, \ldots, \mathcal{G}_n$, respectively.

our goal is to create a syntax-graph $\mathcal{H}$ for h containing $\mathcal{G}_1, \ldots, \mathcal{G}_n$. To achieve this, we construct a second BDD h' as shown in Fig. 2. $G_1, \ldots, G_n$ are newly introduced BDD variables. Since for all m,

$$\begin{aligned} h_{\downarrow \mathcal{G}_1 \wedge \ldots \mathcal{G}_{m-1}} &\equiv \mathcal{G}_m h_{\downarrow \mathcal{G}_1 \wedge \ldots \mathcal{G}_{m-1} \wedge \mathcal{G}_m} \vee \neg \mathcal{G}_m h_{\downarrow \mathcal{G}_1 \wedge \ldots \mathcal{G}_{m-1} \wedge \neg \mathcal{G}_m} \\ &\equiv (G_m h_{\downarrow \mathcal{G}_1 \wedge \ldots \mathcal{G}_{m-1} \wedge \mathcal{G}_m} \vee \neg G_m h_{\downarrow \mathcal{G}_1 \wedge \ldots \mathcal{G}_{m-1} \wedge \neg \mathcal{G}_m})[G_m \leftarrow \mathcal{G}_m] \end{aligned}$$

the newly constructed BDD h' in Fig. 2 is logical equivalent to h if we substitute $G_1, \ldots, G_n$ by the BDDs for $\mathcal{G}_1, \ldots, \mathcal{G}_n$, respectively.

A crucial issue in the construction process is to check the possibility if h can be exclusively constructed out of $\mathcal{G}_1, \ldots, \mathcal{G}_n$ and the logical connectives $\wedge$, $\vee$, and $\neg$. If h has this property, the sub-BDDs $h_{\downarrow f}$ in Fig. 2 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 *hierarchy preserving*.

5 Circuit Abstractions

Assume we are given some combinatorial circuit $\mathcal{F}$ represented as circuit graph. Any circuit $\mathcal{F}'$ obtained from $\mathcal{F}$ by replacing one or more inner nodes by leaves labeled with newly introduced, so called abstraction variables, is called an abstraction of $\mathcal{F}$. More precisely, we define circuit abstractions as follows:

$$\begin{tikzcd}
\mathcal{F},\mathcal{G} \arrow[d, Rightarrow, "\textit{abstraction}"] & & \text{circuit modification for } \mathcal{G} \\
\mathcal{F}',\mathcal{G}' \arrow[rr, Rightarrow, "\textit{rectification}"] & & \text{circuit modification for } \mathcal{G}' \arrow[u, Rightarrow, "\textit{lift}"']
\end{tikzcd}$$

Fig. 3. Circuit-rectifications for abstracted circuits can be lifted to rectifications of the original unabstracted circuits.

Definition 3. *Let $\mathcal{F}, \mathcal{F}'$ be two circuit-graphs. $\mathcal{F}'$ is called an abstraction of $\mathcal{F}$ if there exists a variable substitution σ with*

$$\mathcal{F} = \sigma\mathcal{F}' \tag{6}$$

We now address the question how the rectification method presented in Section 4 can benefit from using circuit abstractions. Assume we are given two combinatorial circuits $\mathcal{F}$ and $\mathcal{G}$ where $\mathcal{F}$ serves as the specification circuit, and $\mathcal{G}$ is supposed to be the implementation. Both circuits are represented with circuit graphs. Further assume that $\mathcal{F} = \sigma\mathcal{F}'$ and $\mathcal{G} = \sigma\mathcal{G}'$ for some variable substitution σ, i.e., $\mathcal{F}'$ and $\mathcal{G}'$ are abstractions of $\mathcal{F}$ and $\mathcal{G}$, respectively. Then, Theorem 1 guarantees that every $\mathcal{F}'$-rectification of $\mathcal{G}'$ can be lifted to an $\mathcal{F}$-rectification of $\mathcal{G}$ (Fig. 3). As the experimental results in Section 7 will show, this can dramatically reduce rectification time. Moreover, it becomes possible to rectify much larger circuits. However, as every abstraction technique, our approach has some drawbacks which will be discussed in detail in Section 5.2.

Theorem 1. *Let σ be a variable substitution and $\mathcal{F}, \mathcal{G}, \mathcal{F}', \mathcal{G}'$ be circuit-graphs with $\mathcal{F} = \sigma\mathcal{F}'$ and $\mathcal{G} = \sigma\mathcal{G}'$ Further assume that $\mathcal{G}'$ is $\mathcal{F}'$-rectifiable at v, i.e., $\mathcal{F}' \equiv \mathcal{G}'[v \leftarrow \mathcal{H}]$ for some circuit-graph $\mathcal{H}$. Then, $\mathcal{G}$ is $\mathcal{F}$-rectifiable at v with circuit-graph $\sigma\mathcal{H}$, i.e.,*

$$\mathcal{F} \equiv \mathcal{G}[v \leftarrow \sigma\mathcal{H}] \tag{7}$$

Proof. By the assumptions of Theorem 1, we get $\mathcal{F} \equiv \sigma\mathcal{F}' \equiv \sigma\left(\mathcal{G}'\left[v \leftarrow \mathcal{H}\right]\right)$. Applying Lemma 1, we get $\sigma\left(\mathcal{G}'\left[v \leftarrow \mathcal{H}\right]\right) \equiv \left(\sigma\mathcal{G}'\right)\left[v \leftarrow \sigma\mathcal{H}\right]$ and rewriting $\sigma\mathcal{G}'$ by $\mathcal{G}$ finally proves $\mathcal{F} \equiv \mathcal{G}\left[v \leftarrow \sigma\mathcal{H}\right]$.

5.1 Computing Abstractions

In this section, we examine two specific kinds of abstractions, i.e., *structural abstractions* (*S-abstractions*) and *logical abstractions* (*L-abstractions*). For computing S-abstractions, we replace structurally identical sub-graphs in $\mathcal{F}$ and $\mathcal{G}$ by a newly introduced abstraction variable. Similarly, L-abstractions prune away logically equivalent circuit parts. Thus, L-abstractions are stronger than S-abstractions. However, S-abstractions are often sufficient, especially when using the method after an optimization step or within an incremental synthesis environment. Then, both circuits usually differ in a very small part of the circuit, only.

Note that – strictly speaking – L-abstractions are not abstractions in the sense of Definition 5. When we construct $\mathcal{F}'$ by substituting logically equivalent sub-graphs in

$\mathcal{F}$ by a common abstraction variable, we cannot always guarantee the existence of a variable substitution σ with $\mathcal{F} = \sigma\mathcal{F}'$ since $\mathcal{F}$ and $\sigma\mathcal{F}'$ are usually not structurally identical. However, $\mathcal{F}$ and $\sigma\mathcal{F}'$ are logically equivalent. Thus, there always exists some circuit graph $\mathcal{F}''$ with $\sigma\mathcal{F}' = \mathcal{F}''$ and $\mathcal{F}'' \equiv \mathcal{F}$. Since $\mathcal{F}$ serves as the specification and is logically equivalent to $\mathcal{F}''$, we can also use $\mathcal{F}''$ as specification instead. Hence, soundness of the abstraction method is not affected.

For computing abstractions for some specification circuit $\mathcal{F}$ and some implementation circuit $\mathcal{G}$, we proceed as follows: In the first step, we determine nodes in $\mathcal{F}$ and $\mathcal{G}$ that are going to be substituted by a common abstraction variable. In case of S-abstractions, we use a hash-table for storing circuit-graphs. For each node, a hash table index is created such that two nodes are mapped to the same index iff their sub-graphs are structurally identical. The hash-table index can be computed in constant time. Thus, index computation of the whole graph can be done linear in the number of graph nodes. In case of L-abstractions, a BDD is computed for each node. Using the BDD reference pointer as index, nodes have the same index if and only if their associated sub-graphs are logically equivalent. For a circuit graph with n nodes, n BDDs have to be computed. Since BDDs can grow exponentially, index computation may also take exponential time.

After computing indices for all graph-nodes, correlated nodes in $\mathcal{F}$ and $\mathcal{G}$ can now be determined for both abstraction types by simply comparing their indices.

In the second step, all nodes with the same index are replaced by a common abstraction variable. For both abstraction types, this can be done in linear time.

When dealing with large circuits, we embed the procedure above in an iterated abstraction algorithm. Using a threshold value τ, we only compute indices for graph nodes having less than τ successor nodes or a BDD representation with less than τ BDD nodes (depending on the computed abstraction type). After abstracting away correlated sub-graphs, the complete abstraction process is repeated until a fix-point is reached. Obviously, the computed results may differ depending on the threshold τ. The bigger the threshold, the more equivalences are usually detected, but the more computation time will be spent within the abstraction algorithm.

5.2 Drawbacks of the Abstraction Method

Performing automatic error correction on abstracted circuits can dramatically decrease computation time and broaden the range of rectifiable circuits. However, it has some drawbacks that are going to be discussed in this Section.

The first noteworthy property is that two equivalent circuits can become inequivalent after abstractions have been computed (see Fig. 4 for an example). This may cause the rectification algorithm to compute unnecessary changes for the implementation circuit. However, the correctness theorem proven in Section 1 guarantees that both circuits remain equivalent after the modifications have been applied. To avoid unnecessary changes to the implementation circuit, equivalence of both designs should be decided beforehand (e.g. by the method presented in [4]).

Another aspect is that the erroneous part of the implementation circuit may be pruned away when computing abstractions. Fig. 5 shows two circuits where the specification is shown on the left and the implementation on the right. Both circuits are not equivalent due to a missing NOT gate at the upper input of component C_1. Both abstraction types result in a circuit where the erroneous position has been abstracted away. Therefore, in this example, all solutions computed for the abstract circuits will rectify the implementation by modifying component C_2. Hence, it is most likely that more modifications are needed than necessary when rectifying the unabstracted circuits directly. However, the experimental results in Section 7 show that especially S-abstractions are very robust in respect to this problem.

A third aspect worth mentioning is that the abstraction algorithm adds new inputs (variables) to the circuit. As BDDs can grow exponentially in the number of variables, this can in principle cause runtime to increase exponentially. Applied to a circuit with n input variables, it is in theory possible that the abstraction algorithm creates a circuit with up to $2^{(2^n)}$ input variables.

However, such an example had to be constructed explicitely and we have observed this phenomenon for none of our example circuits in practice.

6 Rectifying Multiple-Output Circuits

For rectifying multiple output-circuits, there are two possibilities: The first possibility is to rectify every output-signal separately and intersect the solution sets. Then, every solution has to be tested if it also corrects the other signals. If yes, the solution fixes the whole circuit, otherwise it has to be discarded. In practice, this approach finds correct solutions for most multiple output circuits. However, this method is not complete in the

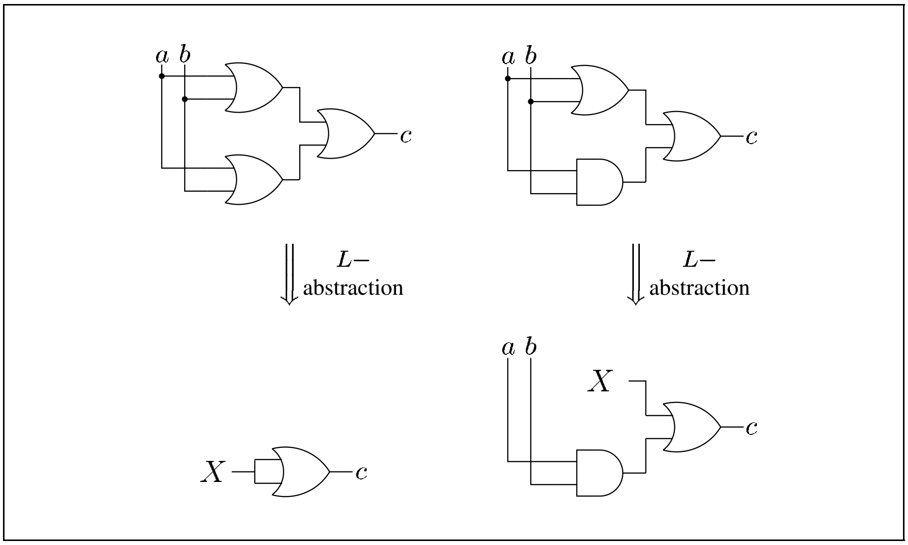


Fig. 4. The two originally equivalent circuits become inequivalent after abstraction. Both unabstracted circuits implement the function $c = a \vee b$.

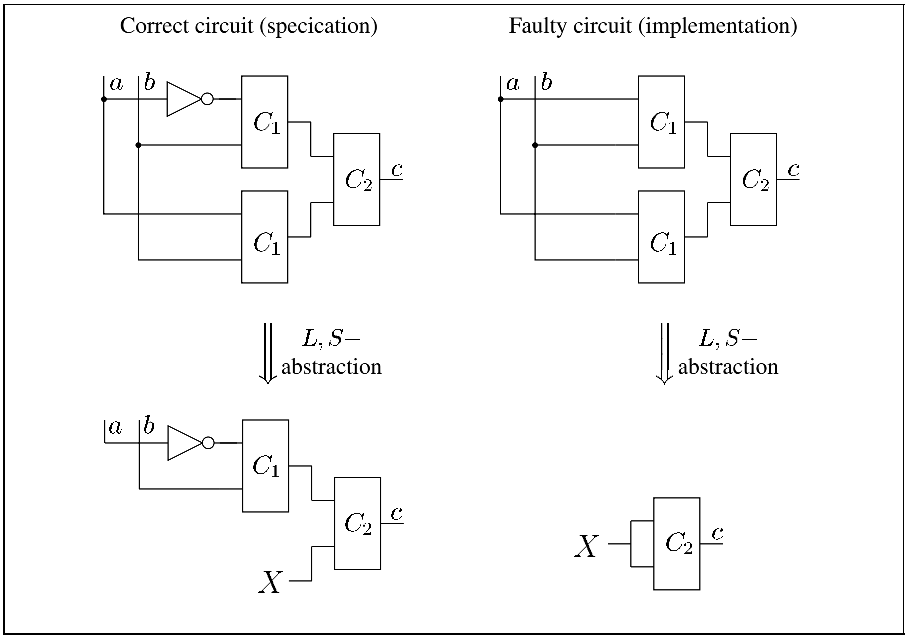

Correct circuit (specication)
Faulty circuit (implementation)
a
b
C_1
C_2
c
$L, S-$ abstraction
X

Fig. 5. In some cases, the erroneous position in the implementation circuit can be pruned away which may lead to unnecessary complex solutions.

sense that it is possible that no rectification is computed that corrects all output signals, even if such a solution exists.

In these cases, we first transform the implementation circuit to a single output circuit as shown in Fig. 6. This transformation basically puts the (abstracted) specification $\mathcal{F}$ and the (abstracted) implementation $\mathcal{G}$ together in one circuit and replaces the old specification by logical true. If $\mathcal{F}$ is not given as a circuit net-list, it has to be synthesized to some net-list equivalent to $\mathcal{F}$.

The transformation assures that whenever the newly created output is logically equivalent to true, all output signals of $\mathcal{F}$ are equivalent to the outputs of $\mathcal{G}$. Obviously, when applying the rectification algorithm to the newly constructed circuit, we have to restrict the solution set to solutions modifying the circuit at nodes belonging to the old implementation circuit.

7 Experimental Results

We have implemented our rectification and abstraction method presented in Section 4 and Section 5 in an equivalence checker and circuit rectifier called **AC/3** [12].

Using **AC/3**, we have evaluated our abstraction method with various benchmark examples, i.e., the Berkeley benchmark circuits [6], and the ISCAS85 benchmark cir-

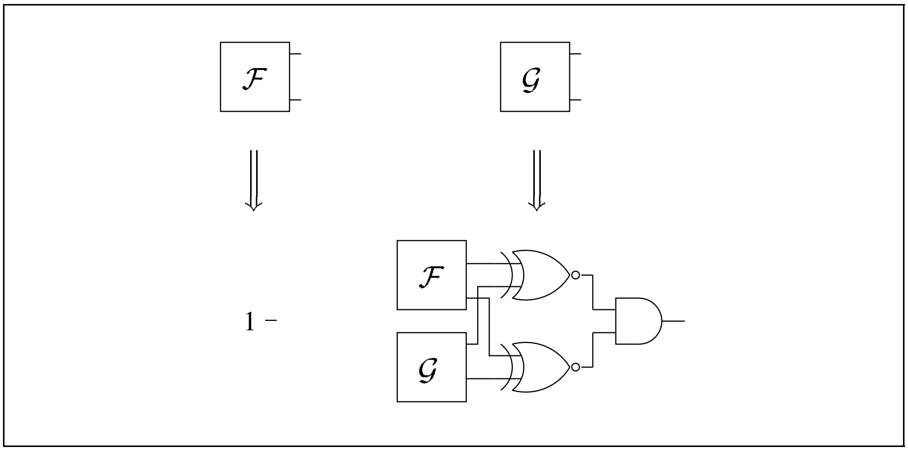


Fig. 6. Transforming $\mathcal{G}$ from a multiple output circuit to a single output circuit. (here, $\mathcal{F}$ and $\mathcal{G}$ have two output signals.)

cuits [7]. In each implementation circuit, we have induced a single error ranging from a missing line, a wrong logical connective, to a missing or double inverter. Computation time has been measured on a Sun Ultra 10 with 128 MB main memory and 300 MHz.

The experimental results for the Berkeley benchmark circuits are shown in the upper half of Table 1 and 2. The first column contains the benchmark's name and the second column is the output signal to which the algorithm has been applied. Without abstraction, all Berkeley benchmark circuits can be rectified. Using structural abstraction, computation time can be reduced drastically for all examples. In 13 out of 15 examples the original error has been found so that the best solution (the solution which requires the minimal number of changes in the implementation circuit) was identical to the best solution computed on the unabstracted circuits. Only for 2 circuits, S-abstraction produced solutions that require more modifications in the implementation circuit. A drawback of logical abstraction seems to be that they are less robust with respect to pruning away the original error position which often results in unnecessary complex solutions (see Section 5.2). However, for the Berkeley Benchmark circuits, in 11 out of 16 examples, the optimal solution has been found.

The experimental results for the ISCAS85 benchmark circuits are shown in the lower half of Table 1 and 2. Without abstraction, 5 out of 10 circuits can be processed successfully. Using abstraction, we have been able to rectify all ISCAS85 circuits. Again, structural abstraction turned out to be more robust in respect to finding the optimal solution. While L-abstraction didn't find the optimal solution in 8 cases, S-abstraction produced the optimal solution in 8 out of 10 examples.

circuit name	no. of inputs	no. of gates	signal name	rectification time	no. of solutions
The Berkeley Benchmark circuits (unabstracted)					
x1dn	27	108	32_out	0.14 sec	10
x9dn	27	89	31_out	0.20 sec	8
x6dn	38	285	42_out	0.89 sec	51
jbp	36	397	87_out	0.58 sec	16
chkn	29	511	539_out	0.83 sec	4
signet	39	240	40_out	2.97 sec	3
in6	33	188	41_out	0.12 sec	9
in7	26	143	31_out	0.21 sec	26
in3	34	302	49_out	0.43 sec	7
in5	24	213	25_out	0.72 sec	47
in4	32	568	33_out	5.10 sec	90
cps	24	936	942_out	4.86 sec	27
bc0	21	952	927_out	8.21 sec	7
The ISCAS85 Benchmark circuits (unabstracted)					
C432	36	160	1355_out	16.08 sec	4
C499	41	202	23_out	25.26 sec	12
C880	60	383	2899_out	0.01 sec	4
C1355	41	546	3882_out	215.78 sec	7
C1908	33	880	5361_out	509.44 sec	48
C2670	233	1193	432_out	> 20 min	—
C3540	50	1669	747_out	> 20 min	—
C5315	178	2307	7754_out	> 20 min	—
C6288	32	2406	6288_out	> 20 min	—
C7552	207	3512	420_out	> 20 min	—

Table 1. Rectification of the Berkeley benchmark circuits and the ISCAS85 benchmark circuits without abstraction.

8 Summary

We have presented a method for localizing and correcting errors in combinatorial circuits. 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 sub-components* and the *computation of circuit corrections*. For both tasks, we have presented efficient solutions based on Boolean decomposition.

Since our method is not structure based, our technique can even be applied in scenarios where the specification is given as a Boolean formula, a truth table, or directly in form of a BDD. When computing circuit corrections, our approach tries to reuse parts of the old circuit in order to minimize the number of modifications and therefore to increase the quality of the computed solutions. Our method is powerful if the error causing elements are concentrated in a comparably small sub-part of the circuit since our algorithm tries to locate the smallest sub-component containing the erroneous parts. This is obviously true, e.g., for all circuits fulfilling the single error assumption.

	Structural abstraction			Logical abstraction		
circuit name	abstraction time	rectification time	no. of solutions	abstraction time	rectification time	no. of solutions
	The Berkeley Benchmark Circuits:					
	Threshold $\tau = \infty$			Threshold $\tau = \infty$		
x1dn	0.23 sec	0.01 sec	4	0.26 sec	0.01 sec	4
x9dn	0.25 sec	$<$ 0.01 sec	7	0.24 sec	0.01 sec	4
x6dn	0.84 sec	0.03 sec	4	0.82 sec	0.03 sec	4
jbp	1.09 sec	0.01 sec	4	1.09 sec	0.02 sec	4
chkn	1.55 sec	0.04 sec	4	1.63 sec	0.03 sec	4
signet	0.73 sec	0.02 sec	3	0.78 sec	0.02 sec	2
in6	0.48 sec	0.02 sec	2	0.5 sec	0.01 sec	2
in7	0.34 sec	0.02 sec	3	0.36 sec	0.02 sec	3
in3	0.79 sec	$<$ 0.01 sec	3	0.79 sec	$<$ 0.01 sec	3
in5	0.60 sec	0.02 sec	5	0.63 sec	0.02 sec	5
in4	1.72 sec	0.16 sec	16	1.79 sec	0.04 sec	8
cps	6.19 sec	0.89 sec	6	10.84 sec	0.02 sec	7
bc0	3.92 sec	0.02 sec	6	3.71 sec	0.02 sec	6
	The ISCAS85 Benchmark Circuits:					
	Threshold $\tau = \infty$			Threshold $\tau = 2000$ (200 for C7552)		
C432	0.44 sec	0.01 sec	4	0.82 sec	0.01 sec	4
C499	0.57 sec	0.01 sec	4	3.01 sec	0.02 sec	4
C880	6.09 sec	0.01 sec	4	8.11 sec	0.01 sec	4
C1355	1.84 sec	0.09 sec	7	8.15 sec	0.03 sec	5
C1908	2.35 sec	0.81 sec	13	9.22 sec	0.71 sec	10
C2670	4.78 sec	0.01 sec	10	14.57 sec	0.01 sec	9
C3540	5.51 sec	0.47 sec	12	18.13 sec	0.45 sec	10
C5315	11.53 sec	0.02 sec	7	22.93 sec	0.01 sec	6
C6288	15.74 sec	0.09 sec	30	74.28 sec	0.07 sec	28
C7552	20.74 sec	0.07 sec	11	24.02 sec	0.1 sec	5

Table 2. Rectification of the Berkeley benchmark circuits and the ISCAS85 benchmark circuits using abstraction. All circuits can be rectified.

To be able to handle large circuits, we have combined the rectification method with circuit abstractions. Two classes of abstractions have been examined: *structural abstractions* and *logical abstractions*. Whereas structural abstractions prune away structurally identical regions of a circuit, logical abstractions remove logically equivalent parts. We have shown correctness of our method by proving that circuit rectifications computed for abstract circuits can be lifted to circuit corrections for the original, unabstracted circuits.

We have implemented the presented methods in the **AC/3** verification tool and evaluated it with the Berkeley benchmark circuits [6] and the ISCAS85 benchmarks [7]. In combination with circuit abstractions, we have been able to rectify all ISCAS85 benchmarks. The experimental results show that together with the abstraction techniques discussed in this paper, our rectification approach is a powerful method for performing automatic error correction of large circuits.

In future, we will extend the rectification method with capabilities to simultaneously rectify circuits at multiple positions. Furthermore, we are currently extending our method to circuits containing tri-states and bus architectures.

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. 158
2. D. Brand. The taming of synthesis. In *International Workshop on Logic Synthesis*, May 1991. 158
3. D. Brand. Incremental synthesis. In *Proceedings of the International Conference on Computer Aided Design*, pages 126–129, 1992. 158, 159
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. 157, 158, 165
5. D. Brand, A. Drumm, S. Kundu, and P. Narain. Incremental synthesis. In *Proceedings of the International Conference on Computer Aided Design*, pages 14–18, 1994. 158, 159
6. 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. 157, 159, 167, 170
7. F. Brglez and H. Fujiwara. A neutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN. In *Int. Symposium on Circuits and Systems, Special Session on ATPG and Fault Simulation*, 1985. 157, 159, 168, 170
8. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986. 157
9. R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992. 157
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. 158
11. A. Gupta. Formal Hardware Verification Methods: A Survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992. 157
12. D. W. Hoffmann and T. Kropf. AC/3 V1.00 - A Tool for Automatic Error Correction of Combinatorial Circuits. Technical Report 5/99, University of Karlsruhe, 1999. available at http://goethe.ira.uka.de/∼hoff. 159, 167
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. 157
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. 158, 159
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. 158
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. 157
17. M. Tomita and H. H. Jiang. An algorithm for locating logic design errors. In *IEEE International Conference of Computer Aided Design (ICCAD)*, 1990. 158

18. 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. 158
19. A. Veneris and I. N. Hajj. Correcting multiple design errors in digital VLSI circuits. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, Orlando, Florida, USA, May 1999. 158
20. 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. 158
21. 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. 158