Preprint from Proceedings of GI/ITG/GMM-Workshop “Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen”, Kaiserslautern, Germany, February 2004

Filter Based Diagnosis for Multiple Design Errors

Christoph Scholl Matthias Büche
Institute of Computer Science
Albert–Ludwigs–University
D 79110 Freiburg im Breisgau, Germany
email: scholl@informatik.uni-freiburg.de bueche@informatik.uni-freiburg.de

Abstract

Both due to the increasing complexity of VLSI circuits and due to the increasing trend to design automation, error diagnosis of digital circuits becomes more and more important. In this paper we present a filter based approach to diagnosis of multiple design errors. Although we present an exact solution to identifying the error location, experimental results show the efficiency of our approach. Our approach is based on the combination of structural arguments and numerous methods differing in their accuracy and complexity. We also present a novel search strategy for error candidates based on recursive partitioning.

1 Introduction

Verification, i.e. the check whether a circuit implementation fulfills its specification, is a crucial task in VLSI CAD. Growing interest in universities and industry has lead to new results and significant advances concerning topics like property checking, state space traversal and combinational equivalence checking [3, 5, 11, 8].

However, developing automated methods for error *detection* is not enough. After obtaining the result that a circuit implementation does not fulfill its specification, circuit designers will have the task of identifying the exact error location and of fixing the error by redesigning the erroneous parts of the design. Due to the complexity of modern designs computer-aided methods for error *diagnosis* are definitely needed. The focus of this paper is to provide automatic methods which help the designer to identify possible error locations in combinational circuits. Our methods work both for single errors and for multiple errors.

Previous work concerning error diagnosis can be roughly divided into two classes:

- testing based error diagnosis and
- synthesis based error diagnosis and rectification.

Testing based methods use simulation of test patterns for error location. Test patterns may be random patterns or generated sets of patterns for some fault model, e.g. the stuck-at-fault model for testing of combinational circuits. Abadir et al. [1] introduced the use of complete test sets for the stuck-at fault model for the detection of design errors, but they did not perform error *location*. Moreover, they introduced simple models for design errors (single gate or single wiring errors) which were used by many authors in the sequel. Testing based error diagnosis methods typically need much CPU time for test vector generation and/or simulation of a large number of input patterns, suffer from a lack of precision in locating the error, or make use of restricted error models, for instance single gate errors or single wiring errors. In many cases they use test patterns not for fault models like the stuck-at model, but dedicated test patterns for design errors in a simple error model [15, 16, 12, 19, 20, 18]. However, simple error models like single gate errors seem not to be realistic for large circuits. Moreover, some approaches such as [19] will even not work for all cases when the replacement of a single gate makes the implementation *functionally* equivalent to the specification – structural similarities are needed as well.[1] Recently Boppana et al. [2] proposed the region-based error model, which is more general. This error model is not restricted to single errors and uses the concept of locality. The region-based error model in connection with a three-valued logic was used for error location in [2], [6], and [14]. However, the methods [2, 6, 14] do not lead to a minimal number of candidate regions for error location due to approximate reasoning.

Synthesis based approaches such as [10, 9, 4] are typically exact solutions in the sense that they are able to restrict the candidate places for error location exactly to those places where the errors can be corrected. Since they use symbolic methods to compute and represent all possible rectifications for the error, they usually show a large memory consumption. E.g. [10] is restricted to single gate errors and has only a very simple concept for searching for the error location: For each possible location there is a check whether the error can be rectified at this place. At least if a single candidate region for the error location is given, methods like the theory of permissible functions [21], which were developed for synthesis and optimization of combinational circuits, can be used for error detection, too. Also in this case all possible rectifications of the error are represented and there is not any sophisticated concept for searching for the error location.

In this paper we present a method of diagnosis for multiple design errors which makes the following contributions:

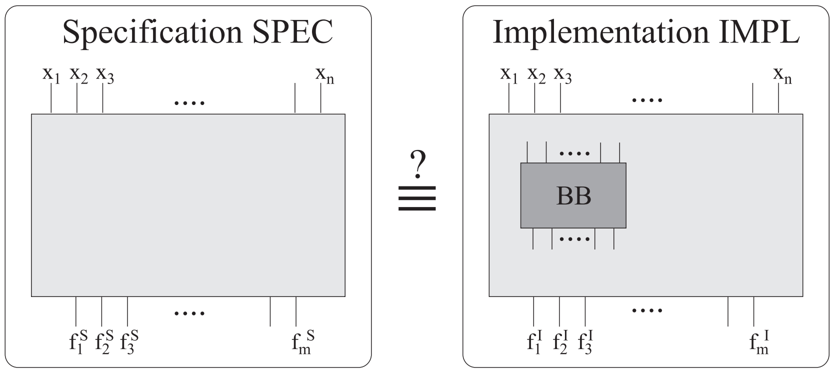
- In contrast to first approaches to error location using a three-valued logic [2, 6, 14] our method is *exact*, i.e., if our method identifies a candidate region for error location, then it is indeed possible to correct the error by changing the design only in this candidate region.
- We intentionally restrict ourselves to exact solutions of the *diagnosis* task. We do not compute a rectification. This leads to a considerable speed-up compared to synthesis methods such as [10, 9, 4, 21]. Because we do not have to represent all possible rectifications for the error, we can make use of optimizations like early quantification [17] when we check whether the error can be corrected within some candidate region.
- Based on the observation that synthesis based methods for error diagnosis typically make a considerable effort even for simple instances we use a filter based approach which applies a series of more and more exact methods for deciding whether the error is restricted to some candidate region. When weaker methods like the simulation for a few input vectors already prove that the errors are not included in or rather are not restricted to a candidate region, we do not need to use more expensive methods. Experimental results prove that simple methods are sufficient to exclude a large number of candidate regions. Here algorithms from [13], which we developed to solve equivalence checks for partial designs, are applied as filters for error diagnosis.
- Moreover, we make use of structure based arguments. We show that in many cases already simple arguments based on the circuit structure are enough to exclude candidate regions for design errors. In addition, structure based arguments are successful in making other filters for error diagnosis more efficient.

We provide efficient solutions to two essential problems which need to be solved for error location:

- Given a candidate region for the error location it has to be checked efficiently whether the error can be corrected only by changing this candidate region.
- Efficient search methods have to be developed which *generate* candidate regions for error location and compute one or all possible error locations using the checks mentioned above.

The paper is structured as follows: In Section 2 basic definitions and notations are given. Section 3 explains how the error location problem is related to the so-called *Black Box Equivalence Checking*

[1]In [19] this comes into play when *ternary* input patterns are used and the output of the specification is X for some input pattern. Then the error is said to be corrected only if the implementation produces X for this input pattern, too. However, due to the X-propagation rules the fulfillment of this requirement may ask for some structural similarities between specification and implementation.



Specification SPEC
Implementation IMPL
x_1 x_2 x_3 .... x_n
x_1 x_2 x_3 .... x_n
$\overset{?}{\equiv}$
BB
f_1^S f_2^S f_3^S .... f_m^S
f_1^I f_2^I f_3^I .... f_m^I

Figure 1: Black Box Equivalence Checking problem.

problem and briefly reviews basic algorithms from [13]. In Section 4 we describe how arguments concerning the circuit structure can be exploited both for deciding Black Box Equivalence Checks and for simplifying Black Box Equivalence Checks. Section 5 explains how the methods from Sections 3 and 4 are used to perform error diagnosis for the region-based error model. Section 6 introduces a novel concept for the search for candidate regions based on recursive partitioning. Finally, we present experimental results in Section 7 and Section 8 summarizes the paper.

2 Preliminaries

Throughout this paper we consider a (combinational) Boolean circuit implementation C^I, which implements a Boolean function $f^I : \{0,1\}^n \to \{0,1\}^m$. As usual, f^I is regarded as a vector $(f^I_1, \ldots, f^I_m)$ of Boolean functions $f^I_j : \{0,1\}^n \to \{0,1\}$ $(1 \leq j \leq m)$. The specification is given by a Boolean function $f^S : \{0,1\}^n \to \{0,1\}^m$. Error diagnosis comes into play when a circuit verifier has detected that $f^I \neq f^S$.

Error diagnosis is supposed to find a subset $eg := \{g_1, \ldots, g_k\}$ of the gates of C^I such that there is a replacement of the gates g_i by gates g'_i ($1 \leq i \leq k$) leading to a Boolean circuit $C^{I'}$ which implements the Boolean function $f^{I'} = f^S$. Since Boolean circuits do not provide a canonical representation for Boolean functions, the set $\{g_1, \ldots, g_k\}$ and the replacing gates $\{g'_1, \ldots, g'_k\}$ are not unique. Thus, our goal is to find a minimal set $\{g_1, \ldots, g_k\}$ or our goal is to find all possible choices for $\{g_1, \ldots, g_k\}$.

Of course, in an erroneous implementation not all outputs need to be erroneous. So we differentiate between two sets of output functions: The set $EO := \{f^I_j \mid f^I_j \neq f^S_j\}$ of erroneous output functions and the set $CO := \{f^I_j \mid f^I_j = f^S_j\}$ of correct output functions. During our structure based preprocessing (see Section 4) we consider for each output i the set of gates TFI_i connected (directly or indirectly) to the gate computing this output.[2] If $f^I_j \in EO$, then TFI_j is called an *'error cone'* and if $f^I_j \in CO$, then TFI_j is called a *'correct cone'*. Of course, cones of different output functions have not to be disjoint, i.e. some gates may be both in an error cone and in a correct cone.

3 Black Box Equivalence Checking

In this section we describe the relation between error diagnosis and the so-called Black Box Equivalence Checking problem. Moreover we give a brief review of different algorithms which we developed in [13] for solving the Black Box Equivalence Checking problem for combinational circuits.

3.1 Black Box Equivalence Checking and Error Diagnosis

The Black Box Equivalence Checking problem is used for checking the correctness of partial designs [6, 13]. Figure 1 illustrates a Black Box Equivalence problem with one Black Box: Suppose we have a complete specification $SPEC$ and a partial implementation $IMPL$. The partial implementation contains a so-called Black Box (which is a part of the circuit that is not finished, not known or abstracted away). The Black Box Equivalence Checking problem asks if there is a replacement of the Black Box by some circuit which makes the overall implementation correct, i.e. which makes the

[2]This set of gates is usually denoted as 'transitive fan-in' or as 'Cone-of-Influence' (COI) of output i.

implementation $IMPL$ functionally equivalent to the specification $SPEC$.

The relation between error diagnosis and the Black Box Equivalence Checking problem is straightforward: If a complete implementation, which was proven to be incorrect, and a candidate region for the error location are given, then it can be decided using Black Box Equivalence Checking whether it is possible to correct the error by changing gates only in this candidate region. The gates of the candidate region are simply combined into a Black Box and Black Box Equivalence Checking is applied. If Black Box Equivalence Checking leads to a positive result, then we know that it is possible to find a Black Box implementation which makes the overall implementation correct. This means that the assumption about the error location was correct. On the other hand, if Black Box Equivalence Checking leads to a negative result, then the error can not be corrected just by changing gates in the given candidate region, i.e., there has to be an error also outside the Black Box.

3.2 Algorithms for Black Box Equivalence Checking

We briefly review basic algorithms which can be used for an approximate solution to the Black Box Equivalence Checking problem [13]. The algorithms need different amounts of resources (space and time) and differ from their accuracy: They range from non-symbolic simulations using a ternary $(0, 1, X)$-logic for approximating the solution up to an exact solution to the problem. In this context an algorithm is called approximate, if it is able to find errors in the partial implementation, but it does not necessarily detect all errors. However, the algorithm has to be sound in the sense that it never reports an error, if there is a Black Box implementation which makes the overall implementation correct. Thus, with respect to error diagnosis an approximate algorithm is only able to definitely exclude candidate regions for errors (when it produces a negative result for the candidate region as a Black Box), but it can never guarantee that an implementation can be corrected only by changing gates from some candidate region, since a positive answer of the algorithm can be due to its approximative character.

However, it makes sense to apply approximate algorithms all the same, since they provide an efficient method to exclude candidate regions.

So approximate algorithms can give two different answers: ‘there has to be an error outside the Black Box’ and ‘all errors *may* be inside the Black Box’. Exact algorithms give either the answer ‘there has to be an error outside the Black Box’ or ‘*all* errors are *definitely* inside the Black Box’.

For checking whether design errors are included in some candidate region we use the following series of algorithms for Black Box Equivalence Checking:

3.2.1 Non-symbolic simulation using a ternary $0, 1, X$-logic

The first check is based on non-symbolic simulation. First of all, a set of ‘erroneous input vectors’ from $\{0, 1\}^n$ is identified for which the results at the outputs of implementation and specification are different. For each candidate region a Black Box is introduced. The outputs of the Black Box are assigned to the value X (for any unknown value) and erroneous input vectors are simulated using the ternary $0, 1, X$-logic [2]. Whenever we observe a 0 at some output of the implementation and a 1 at the corresponding output of the specification (or vice versa), we can conclude that this difference will exist for all Black Box implementations, since it does not depend on the X-values at the Black Box outputs. Thus, this region is no longer a candidate region for the error correction.

3.2.2 Symbolic simulation based on $0, 1, X$-logic

Since non-symbolic simulation usually considers only a subset of the erroneous input vectors, a more exact algorithm can be obtained using symbolic simulation. In [13] a modified symbolic simulation is defined which computes for each output j of the implementation C^I a BDD representation of the Boolean function $\widetilde{f}_j^I(x_1, \ldots, x_n, Z)$ with

$$\widetilde{f}_j^I|_{x_1=\epsilon_1,\ldots,x_n=\epsilon_n} = \begin{cases} 1, \text{ if non-symbolic } (0,1,X)\text{-simulation with input } (\epsilon_1,\ldots,\epsilon_n) \text{ produces } 1 \\ 0, \text{ if non-symbolic } (0,1,X)\text{-simulation with input } (\epsilon_1,\ldots,\epsilon_n) \text{ produces } 0 \\ Z, \text{ if non-symbolic } (0,1,X)\text{-simulation with input } (\epsilon_1,\ldots,\epsilon_n) \text{ produces X} \end{cases}$$

An error outside the Black Boxes is detected if there is an output $1 \leq j \leq m$ with $\forall Z \left(\widetilde{f}_j^I \oplus f_j^S\right) \neq 0$.

3.2.3 Symbolic Z_i-simulation, local check

Again, a more exact (but computationally more expensive) result is obtained, if unknown values at the Black Box outputs are not modelled by a single variable Z, but by different variables Z_i for each

Black Box output. In [13] for each primary output j of the implementation a function $\widehat{f_j^I}$ is computed which depends on the primary input variables $x_1, \ldots, x_n$ and l variables $Z_1, \ldots, Z_l$ (supposed there are l outputs of the Black Boxes). A similar check for errors is performed for each output j as in the case of symbolic simulation based on $(0, 1, X)$-logic: An error outside the Black Boxes is detected if there is an output $1 \leq j \leq m$ with $\forall Z_1 \ldots \forall Z_l \left(\widehat{f_j^I} \oplus f_j^S\right) \neq 0$.

3.2.4 Symbolic Z_i-simulation, output exact check

The next check is based on the same symbolic representation of functions $\widehat{f_j^I}$ as the previous one, but it takes correlations of correctness conditions for different outputs into account. It detects an error already when the correctness conditions for different outputs can not be fulfilled by all outputs $1 \leq j \leq m$ *at the same time*. From Section 3.2.3 a local correctness condition $cond_j = \widehat{f_j^I} \equiv f_j^S$ is derived for each output and an error outside the Black Boxes is detected iff $\forall Z_1 \ldots \forall Z_l \left(\bigvee_{j=1}^m \overline{cond_j}\right) \neq 0$.

3.2.5 Symbolic Z_i-simulation, input exact check

The last check takes into account that in the general case a Black Box implementation cannot compute its output functions simply based on the primary inputs. The Black Box output functions are computed based on the input functions provided to the Black Box inputs by the implementation C^I. So the Black Box outputs cannot compute arbitrary functions in terms of primary inputs and this fact is exploited to tighten the previous check once more. Assume that there is only one Black Box with input variables $Y_1, \ldots, Y_k$ and output variables $Z_1, \ldots, Z_l$ and assume that the relationship between the values at the primary inputs $x_1, \ldots, x_n$ of C^I and the input variables of the Black Box $Y_1, \ldots, Y_k$ is given by the Boolean function $H(x_1, \ldots, x_n, Y_1, \ldots, Y_k)$.[3] Then the so-called input exact check reports an error outside the Black Box iff

$$\forall Z_1 \ldots \forall Z_l \exists x_1 \ldots \exists x_n \left[H(x_1 \ldots x_n, Y_1, \ldots, Y_k) \cdot (\bigvee_{j=1}^{m} \overline{cond_j}(x_1, \ldots, x_n, Z_1, \ldots, Z_l)) \right] \neq 0.$$

Note that [13] proves this check to be exact, i.e., if it does not report an error outside the Black Box, then it is possible to find an implementation of the Black Box which makes the overall implementation correct. In [13] also the case of multiple Black Boxes is considered.

4 Exploiting Structure Based Arguments

The algorithms described in Section 3.2 provide a series of more and more exact checks for the Black Box Equivalence Checking problem. In this section we present how to improve this series by simple arguments on the structure of circuit C^I. Structure based arguments are used for a simple check excluding candidate regions for error location and, moreover, they are used to make the checks from Section 3.2 more efficient.

4.1 A simple structure based check for the Black Box Equivalence Checking problem

First of all, we make use of error cones as defined in Section 2. It is straightforward that in some candidate region for the error location not all errors can be included, if its intersection with some error cone is empty. Of course, this candidate region cannot correct the error for the primary output corresponding to this error cone. So we can restrict ourselves to candidate regions whose intersection with all error cones is not empty. This simple check is applied already before non-symbolic $0, 1, X$-simulation.

If we restrict ourselves to a single error model, i.e., if we assume that there is only one erroneous gate in the circuit C^I, then this check can be tightened: Under this assumption it is a necessary condition for some candidate region to contain the error that there is at least one gate in this region which is in the intersection of all error cones. (Due to the single error assumption the replacement of the erroneous gate has to correct all erroneous outputs.) Moreover, in this case candidate regions can be made smaller by a restriction to the gates in the intersection of all error cones.

[3] $H(x_1, \ldots, x_n, Y_1, \ldots, Y_k) = 1$ if and only if the simulation of the implementation C^I with input vector $(x_1, \ldots, x_n)$ produces the value $(Y_1, \ldots, Y_m)$ at the Black Box inputs.

4.2 Simplifying checks using structure based arguments

Structure based arguments can be used to make the checks from Section 3.2 more efficient (especially the output and the input exact check). These checks can be simplified, if the correct cones are removed from C^I (and from the specification). Experimental results (see Section 7) have shown that the sizes of the BDDs, the number of variables and the complexity of operations (in particular quantifications with respect to a smaller number of variables) are reduced to a great extend by this simple measure.

5 Diagnosis for Region-Based Model

In this section we describe how we use the checks from Sections 4 and 3.2 in connection with the region-based error model introduced by Boppana et al. [2]. In this error model Boppana et al. make the assumption of locality of errors. Typically a region consists of a gate in the circuit and all its surrounding gates. The radius of a region specifies the actual size of the region. E.g. a region of radius 0 around a gate consists only of the gate itself. A region of radius 1 consists of the gate and all its immediate successors and predecessors in the circuit. Since each gate can be used as the center gate of a region, there will be as many regions of a fixed radius as the number of gates in the circuit. However, the regions will be overlapping with each other when their radii are greater than 0. The region-based error model allows multiple errors in the regions, but all errors will be restricted to a region of a given radius.

According to the region-based error model we start with a number of regions of fixed radius which is equal to the number of gates in the circuit C^I. To cut down the number of candidate regions we do not use a single algorithm, but we use the series of algorithms given in Sections 4 and 3.2.

Thus, we begin with simple structure based arguments and apply them to all candidate regions to reduce their number. Then we perform a non-symbolic event-driven $(0,1,X)$-simulation. Note that during the non-symbolic event-driven $(0,1,X)$-simulation we pass through all candidate regions for the simulation of one erroneous input vector before we proceed to the next erroneous input vector. In this way, the event-driven simulation can reuse parts of the simulation results for the same erroneous vector when simulation is applied for different locations of the candidate region. After non-symbolic $(0,1,X)$-simulation we apply the symbolic $(0,1,X)$-based simulation to the remaining candidates, and so on.

Using this filter-based approach we avoid shortcomings of synthesis based methods for error diagnosis: We do not spend a lot of run time for instances which are not really hard. Instances which can be solved using weaker methods are not presented to the more powerful and more expensive checks. Moreover, the number of instances presented to the more powerful checks becomes smaller and smaller. And furthermore, we are able to profit from optimization techniques like early quantification [17], since our checks just produce a Boolean output instead of providing the set of all possible rectifications.

6 Recursive Partitioning Approach

In this section we present a concept for error diagnosis in the case that no error model exists. In particular, we do not make any assumption about locality of errors as in the previous section. The concept is based on recursive partitioning and the series of more and more exact algorithms given in Sections 4 and 3.2.

A sketch of the algorithm is given in Figure 2. The algorithm is invoked with a parameter $regions\_to\_partition$ which is a partition of a subset of all gates in C^I. At the beginning $regions\_to\_partition$ = $\{C^I\}$. (Here C^I means the set of gates in the erroneous implementation C^I). The result of the procedure is a set of gates in C^I which can be replaced to correct the implementation.

We always maintain two invariants (lines 2–4): Invariant 1 (line 2) says that the errors can be corrected by reimplementing the region which consists of the union of all sets in $regions\_to\_partition$. At the beginning, when $regions\_to\_partition$ = $\{C^I\}$ this is certainly true.[4] Moreover, we maintain Invariant 2 (lines 3, 4) which says that we cannot omit any element of $regions\_to\_partition$ *as a whole* without destroying this property. Of cause, this property holds at the beginning, too, since omitting C^I in $regions\_to\_partition$ = $\{C^I\}$ removes all gates of the erroneous implementation.

During the execution of the algorithm we reduce the size of $\bigcup_{region \in regions\_to\_partition} region$ step by

[4]We assume that the implementation C^I has the same number of inputs and outputs as the specification.

```
 1 set_of_gates **function** *recursive_partitioning*(set_of_sets_of_gates $regions\_to\_partition$)
 2 // Invariant 1: Errors can be corrected inside $\bigcup_{region \in regions\_to\_partition} region$
 3 // Invariant 2: There is no $region' \in regions\_to\_partition$ such that the error can be corrected inside
 4 //               $\bigcup_{region \in regions\_to\_partition \setminus \{region'\}} region$
 5 $curr\_region\_to\_partition$ := *largest_element_of*($regions\_to\_partition$)
 6 **if** $number\_of\_gates(curr\_region\_to\_partition) = 1$ **then return** $\bigcup_{region \in regions\_to\_partition} region$ **fi**;
 7 $regions\_to\_partition := regions\_to\_partition \setminus curr\_region\_to\_partition$;
 8 $(part_0, part_1) := partition(curr\_region\_to\_partition)$;
 9 $black\_box_0 := part_0 \cup \bigcup_{region \in regions\_to\_partition} region$;
10 $black\_box_1 := part_1 \cup \bigcup_{region \in regions\_to\_partition} region$;
11 $(result_0, result_1) := check(black\_box_0, black\_box_1)$;
12 **if** $result_i = error\_definitely\_inside\_bb$
13   **then**
14        **return** *recursive_partitioning*($regions\_to\_partition \cup \{part_i\}$);
15 **fi**
16 **if** $result_0 = result_1 = error\_outside\_bb$
17   **then**
18        Retry partitioning and checking, i.e. goto line 8 until upper limit for retries is reached
19        **if** upper limit of retries is reached
20          **then**
21               **return** *recursive_partitioning*($regions\_to\_partition \cup \{part_0, part_1\}$);
22        **fi**
23 **fi**
```

Figure 2: Pseudo code for *recursive_partitioning*.

step until no further reduction can be accomplished and finally the computed set of erroneous gates is equal to $\bigcup_{region \in regions\_to\_partition} region$ for the resulting $regions\_to\_partition$. At the end of the algorithm $regions\_to\_partition$ consists only of singletons. In order to achieve this goal, we use the idea of dividing single elements $region$ of $regions\_to\_partition$ into two parts such that we can remove one of these two parts of $region$ in $regions\_to\_partition$.

We give a sketch the algorithm by describing the execution starting with $regions\_to\_partition = \{C^I\}$. Then in line 8 the set of gates of $curr\_region\_to\_partition = C^I$ is divided into two (about equal sized) sets $part_0$ and $part_1$.[5] After that, we try two candidate sets for error location: We check the hypothesis that the errors are completely contained in $part_0$ ($black\_box_0 = part_0$, line 9) and the hypothesis that the errors are in $part_1$ ($black\_box_1 = part_1$, line 10). These checks are performed by the function $check$ which implements exactly the series of checks described in Section 5, now applied to exactly two candidate regions $black\_box_0$ and $black\_box_1$.

Remember that approximate checks can provide exactly two answers: 'there has to be an error outside the Black Box' ($result_i = error\_outside\_bb$) and 'the errors *may* be inside the Black Box' ($result_i = error\_maybe\_inside\_bb$). Exact algorithms can provide either the answer 'there has to be an error outside the Black Box' ($result_i = error\_outside\_bb$) or '*all* errors are *definitely* inside the Black Box' ($result_i = error\_definitely\_inside\_bb$).

If the series of checks provides the result $result_i = error\_definitely\_inside\_bb$ for at least one $i \in \{0, 1\}$ (lines 12–15), then we know that the error can be corrected in part i of the circuit only and thus we apply recursive partitioning to this part of the circuit to obtain even more information on the error location (by *recursive_partitioning*($\{part_i\}$) (line 14) in this case). If we obtain the result '$error\_outside\_bb$' for *both* parts of the circuit (line 16), then we know that our partitioning in line 8 did not succeed in separating the error locations from the remainder of the circuit. So we retry partitioning and checking for a number of attempts (line 18).[6] If an upper limit for retries has been reached (line 19), we give up the attempt of separating the error locations by partitioning in line 8 and we have to use recursive partitioning to solve this task. To do so, we invoke the procedure by *recursive_partitioning*($\{part_0, part_1\}$) (line 21). During this recursive call, the larger part, say $part_0$, e.g., is partitioned into two parts $part_{0,0}$ and $part_{0,1}$ and we check whether the error locations are

[5]In a prototype implementation of the algorithm we use METIS [7] to perform this partitioning.

[6]We compute another partition by invoking METIS with different parameters.

Circuit	#erroneous vectors	#regions	struct. check 1	non-symb. $0,1,X$-sim.	struct. check 2	symb. $0,1,X$-sim.	Z_i-sim., local	Z_i-sim., out. ex.	Z_i-sim., inp. ex.
C432	90.90	216	130.60	11.60	11.60	11.60	5.90	5.30	1.70
C499	134.30	246	177.70	15.10	15.10	15.10	11.80	10.60	4.30
C880	91.20	409	93.10	7.10	7.10	7.00	6.80	6.80	2.50
C1355	119.60	558	398.00	23.20	23.20	23.00	17.60	17.60	3.90
C1908	298.20	1056	581.30	34.10	34.10	34.10	19.90	18.20	10.10
C2670	120.0	1460	124.90	24.80	24.50	22.30	16.70	16.40	4.40
C3540	265.90	1981	662.60	24.00	23.90	23.10	16.60	15.90	4.30
C5315	254.30	2963	242.60	25.90	25.90	20.10	18.20	17.30	3.50
C6288	60.0	2416	1156.60	246.40	-	-	-	-	-
C7552	369.50	4040	752.20	95.90	95.90	94.60	67.80	66.80	6.30

Table 1: Number of candidate regions after application of different filters.

restricted to $part_{0,0} \cup part_1$ or to $part_{0,1} \cup part_1$, i.e., the gates in $part_1$ are fixed in the Black Box.
In the general case, the procedure is called by *recursive_partitioning*($regions\_to\_partition$) where $regions\_to\_partition$ is a set of sets of gates. As mentioned above we always maintain invariants 1 and 2. The sets in $regions\_to\_partition$ are candidates for further partitioning and we select the largest set for further partitioning (line 5). The other sets are fixed in the Black Box in both variants $black\_box_0$ and $black\_box_1$ of Black Boxes (lines 9, 10). Partitioning finishes exactly if all sets in $regions\_to\_partition$ contain only one gate (lines 5, 6). In this case we have achieved our goal, since Invariant 2 says that we cannot omit any gate without destroying the property that the errors can be corrected by reimplementing the region defined by the union of gates in $\bigcup_{region \in regions\_to\_partition} region$.
In particular for large circuits, the check of line 11 needs to be discussed in more detail. According to Section 5 and the description of procedure *recursive_partitioning* given above we apply the series of more and more exact checks until we obtain the result '$error\_outside\_bb$' for both candidates (probably by an approximate algorithm) or until the exact algorithm produces the result $error\_definitely\_inside\_bb$ for at least one of the two candidates. For instances where an exact solution is too expensive it seems to be advisable to change the algorithm such that we can also get into line 14 in case '$result_i = error\_maybe\_inside\_bb$' instead of '$result_i = error\_definitely\_inside\_bb$', when we assume that the probability for error correction inside $black\_box_i$ is high enough. (This may especially be the case when we obtained the result '$error\_outside\_bb$' for the other Black Box.) However, we then have to provide a backtracking mechanism for the case that we realize later on during the algorithm that this assumption was incorrect.
The situation becomes much easier when we can assume single errors: If only one gate is erroneous, then the result '$result_i = error\_outside\_bb$' for one of the Black Boxes (possibly obtained by some 'weak', approximate algorithm) definitely implies '$result_{1-i} = error\_definitely\_inside\_bb$' for the other Black Box. However, as already mentioned in Section 1, the single error model does not seem to be realistic, since we do not expect to observe many instances that contain exactly one erroneous gate. But note that – in contrast to other approaches such as [19] – our algorithms do not imply a restriction of the notion of 'single gates' to pure and gates, or gates or inverters. If we consider the single error model not for single gates in the classical sense but for larger functional blocks, then it can make sense all the same. Applying the search procedure first at the level of functional blocks with a single error model will restrict the error location to some functional block. Later on, the candidate region for error location may be further narrowed down by dismissing the single error model.

7 Experimental Results

We performed experiments to locate errors both under the assumption of the region-based error model and using the recursive partitioning approach. First results for the recursive partitioning approach are promising, but due to lack of space we only present results for the region based error model here. All experiments were performed under *Debian Linux 4.0* on an AMD XP 1600+ machine with 1 GB main memory. We applied the method to all benchmarks from the ISCAS85 benchmark set. To inject errors we selected a random gate, considered a region of radius 1 surrounding this gate and changed gates of this region with a probability of 70%. The error type was also selected randomly between several choices: We added/removed an inverter for an input or output signal of the gate, changed the

Circuit	struct. check 1	non-symb. $0, 1, X$-sim.	BDD spec.	BDD impl.	struct. check 2	symb. $0, 1, X$-sim.	Z_i-sim., local check	Z_i-sim., output exact	Z_i-sim., input exact
C432	< 0.01	0.01	0.18	0.09	< 0.01	0.01	0.63	0.70	0.41
C499	< 0.01	0.02	7.37	7.88	< 0.01	2.01	3.91	37.98	22.75
C880	< 0.01	< 0.01	0.62	0.15	< 0.01	0.05	0.30	16.74	12.03
C1355	< 0.01	0.34	5.21	6.76	< 0.01	0.58	24.29	30.27	35.76
C1908	< 0.01	5.33	1.93	0.52	< 0.01	1.02	2.63	3.48	2.02
C2670	< 0.01	1.75	3.19	1.03	< 0.01	1.06	2.07	1.47	1.20
C3540	< 0.01	4.51	13.09	5.89	< 0.01	13.03	155.14	34.56	24.09
C5315	< 0.01	9.27	1.74	0.53	< 0.01	0.01	0.14	0.65	0.84
C6288	< 0.01	49.56	-	-	-	-	-	-	-
C7552	< 0.01	27.64	8.05	1.50	< 0.01	0.04	10.20	4.77	2.69

Table 2: CPU times in CPU seconds.

type of the gate (and_2 to or_2 or or_2 to and_2) or removed an input line from an and or or gate. All experiments are an average of 10 random error insertions. We used a limit of 3 CPU hours for each experiment.

Tables 1 and 2 show the results of the method presented in Section 5 under the assumption of a region-based error of radius 1. Column 1 of table gives the name of the circuit, column 2 the average number of erroneous input vectors used for non-symbolic $(0, 1, X)$-simulation of the various circuits. Erroneous input vectors were obtained by random simulation. The remaining columns show the successive decrease of the number of candidate regions for errors. Column 3 gives the number of different regions of radius 1, respectively. This number is reduced by a sequence of more and more exact checks which also increase with respect to their need for computational resources. Column 4 shows that a large number of candidate regions can already be removed based on very simple structural arguments (see Section 4). Again, many candidate regions can be removed by a simulation based method, see Column 5. After simulating the designs with erroneous input vectors the number of candidate regions is further reduced using symbolic methods. Since simulation can not always identify all erroneous outputs, the structure based check is repeated after computing BDDs for specification and implementation, i.e. after obtaining exact knowledge on the erroneous outputs. Since we potentially identify new outputs as erroneous, the results of the structural check of Section 4 may improve (see column 6). Column 7 shows the results after symbolic simulation based on $(0, 1, X)$-logic. Columns 8, 9, and 10 show the results of the checks based on Z_i-simulation combined with local check, output exact check and input exact check, respectively. It can be observed that for this set of experiments – apart from an initial decrease of the number of candidate regions by structural arguments and by non-symbolic simulation – the most obvious decrease of the number of candidate regions occurs during the local check of Z_i-simulation and during the final input exact check. Note that the final input exact check is exact. Thus the errors can indeed be corrected by changes in several candidate regions (on the average between 1.7 and 10.1 regions for this set of experiments). C6288 was the only circuit where the experiment had to be aborted due to the limit on CPU time.[7]

Table 2 gives some more information on CPU times. The labelling of columns is similar to Table 1. Columns 4 and 5 show CPU times in seconds for BDD construction of specification and implementation, respectively. All other columns give CPU times for the different checks already described for Table 1. The table shows the surprising result that all problems except for the multiplier could be solved within a few seconds. The longest run time occurred for C3540 and amounts to about 250 seconds altogether – during this time the number of candidate regions could be reduced to the *exact* minimum. Even for the more complex checks for solving the Black Box Equivalence Checking problem (columns 9, 10) the requirements for run times were moderate and did not substantially exceed the run times for simpler checks . Most surprisingly, also the last and exact check (input exact check, see column 10) could be completed within a few seconds for the remaining candidates.

The first reason for this observation lies in the fact that the more powerful checks are used for fewer candidate regions: They are used only for candidate regions which were not already excluded by weaker checks. We do not use powerful checks for simple problems.

[7]After structural checks and non-symbolic simulation the BDD construction for the multiplier could not be performed with available resources.

The second reason lies in the fact that these checks have been optimized: Since we restrict ourselves to error *location*, we do not need to represent the set of all possible rectifications of the error. Our checks only produce a Boolean information. This makes it possible to use techniques like early quantification [17] to optimize the checks. Moreover, the checks could be simplified to a great extend by using simple structure based arguments as described in Section 4.2.

8 Conclusions and future work

We have presented a filter based approach to error diagnosis. This approach uses a series of more and more exact filters in order to exclude candidate regions step by step. In this way we do not waste run time by processing simple instances with powerful methods. Moreover, a concept for error location without any error model using a recursive partitioning approach was presented. Experimental results under the assumption of the region-based error model were given to prove the efficiency of our approach.

For the future we plan to make use also of SAT based checks for error diagnosis. To do so, we intend to improve existing approaches [6] and to develop more exact checks based on SAT-engines. Another interesting question is how the methods can be extended to perform error diagnosis for sequential circuits.

References

[1] M.S. Abadir, J. Ferguson, and T.E. Kirkland. Logic design verification via test generation. *IEEE Trans. on CAD*, 7(1):138–148, 1988.

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

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

[4] P.-Y. Chung, Y.-M. Wang, and I.N. Hajj. Diagnosis and correction of logic design errors in digital circuits. In *Design Automation Conf.*, pages 503–508, 1993.

[5] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373. Springer Verlag, 1989.

[6] A. Jain, V. Boppana, R. Mukherjee, J. Jain, M. Fujita, and M. Hsiao. Testing, verification, and diagnosis in the presence of unknowns. In *VLSI Test Symp.*, pages 263–269, 2000.

[7] G. Karypis and V. Kumar. *METIS: A Graph Partitioning Package*. University of Minnesota, 1998. Also available at http://www.cs.umn.edu/∼karypis.

[8] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Design Automation Conf.*, pages 263–268, 1997.

[9] H.-T. Liaw, J.-H. Tsaih, and C.-S. Lin. Efficient automatc diagnosis of digital circuits. In *Int'l Conf. on CAD*, pages 464–467, 1990.

[10] J.C. Madre, O. Coudert, and J.P. Billon. Automating the diagnosis and rectification of design errors with PRIAM. In *Int'l Conf. on CAD*, pages 30–33, 1989.

[11] I.-H. Moon, J. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: The question in image computation. In *Design Automation Conf.*, pages 23–28, 2000.

[12] I. Pomeranz and S.M. Reddy. On diagnosis and correction of design errors. In *Int'l Conf. on CAD*, pages 500–507, 1993.

[13] C. Scholl and B. Becker. Checking equivalence for partial implementations. In *Design Automation Conf.*, pages 238–243, 2001.

[14] N. Sridhar and M.S. Hsiao. On eficient error diagnosis of digital circuits. In *Int'l Test Conf.*, pages 678–687, 2001.

[15] M. Tomita, H.-H. Jiang, T. Yamamoto, and Y. Hayashi. An algorithm for locating desing errors. In *Int'l Conf. on CAD*, pages 468–471, 1990.

[16] M. Tomita, T. Yamamoto, F. Sumikawa, and K. Hirano. Rectification of multiple logic design errors in multiple output circuits. In *Design Automation Conf.*, pages 212–217, 1994.

[17] H. Touati, H. Savoj, B. Lin, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Implicit enumeration of finite state machines using BDDs. In *Int'l Conf. on CAD*, pages 130–133, 1990.

[18] A. Veneris and I.N. Hajj. A fast algorithm for locating and correcting simple design errors in VLSI digital circuits. In *Great Lakes Symp. VLSI*, pages 45–50, 1997.

[19] A. Wahba and D. Borrione. A method for automatic design error location and correction in combinational logic circuits. *Jour. of Electronic Testing: Theory and Applications*, 8(2):113–127, 1996.

[20] A.M. Wahba and D. Borrione. Connection error location and correction in combinational circuits. In *European Design & Test Conf.*, pages 235–241, 1997.

[21] Y. Watanabe, L.M. Guerra, and R.K. Brayton. Permissible functions for multioutput components in combinational logic optimization. *IEEE Trans. on CAD*, 15:732–744, 1996.