Checking Equivalence for Circuits Containing Incompletely Specified Boxes

Christoph Scholl Bernd Becker
Institute of Computer Science
Albert–Ludwigs–University
D 79110 Freiburg im Breisgau, Germany
scholl@informatik.uni-freiburg.de, becker@informatik.uni-freiburg.de

Abstract

We consider the problem of checking whether an implementation which contains parts with incomplete information is equivalent to a given full specification. We study implementations which are not completely specified, but contain boxes which are associated with incompletely specified functions (called Incompletely Specified Boxes or IS–Boxes).

After motivating the use of implementations with Incompletely Specified Boxes we define our notion of equivalence for this kind of implementations and present a method to solve the problem.

A series of experimental results demonstrates the effectiveness and feasibility of the methods presented.

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 [5, 8, 21, 17, 6, 23, 16].

For the purpose of this paper combinational equivalence checking is of particular interest. Here, the task is to check whether the Boolean functions corresponding to the specification and the implementation are the same. Besides functional validation by the application of test patterns, mainly two approaches are used to perform the equivalence check: One possibility is to translate implementation and specification into one Boolean formula which is satisfiable if and only if implementation and specification do not realize the same Boolean function [29, 19, 11]. As an alternative, implementation and specification can be transformed into a canonical form such that the equivalence check reduces to a check whether the canonical representations of implementation and specification are the same. BDDs [1] and Word-level Decision Diagrams such as *BMDs [4], HDDs [7] or K*BMDs [10] are popular choices for such canonical forms. Recent approaches integrate the use of BDDs and SAT solvers to combine advantages of both methods [13, 6, 23, 16]. When specification and implementation are structurally similar, correspondences between internal nodes can be used to simplify the verification problem [15, 24, 20, 17, 6, 23].

Recently, the problem of 'Black Box Equivalence Checking', which occurs when the specification is known, but only parts of the implementation are finished or known, has been addressed [14, 12, 26]. Parts of the implementation which are not finished or known are put into 'Black Boxes'. An error is found in an implementation with Black Boxes, if the implementation differs from the specification *for all possible substitutions of the Black Boxes*, i.e. there exists no completion of the partial implementation that makes it equivalent to the specification.

There are several motivations for considering the Black Box Equivalence Checking problem: One application is equivalence checking in early stages of the design, when a partial implementation is not yet finished, a second application is the 'abstraction from difficult parts' in a (finished) implementation, and a third application consists in error diagnosis, where candidate regions for design errors are defined to be Black Boxes resulting in a Black Box Equivalence Checking problem.

In this paper we look into a related problem, which also handles implementations with incomplete information: Here we assume boxes, which are not completely unknown like Black Boxes, but rather implement an *incompletely specified function*. We call boxes of this kind *Incompletely Specified Boxes* or *IS–Boxes*. (For small examples of implementations with IS–Boxes see Figure 1. The IS–Box realizes the function $\overline{i_1^{(1)}} + \overline{i_2^{(1)}}$ with a don't care for input vector $(0, 0)$. The corresponding specification for the whole circuit is the function $f^S = x_1 \oplus x_2$.)

In the following paragraphs we describe scenarios where it is possible to profit from our approach:

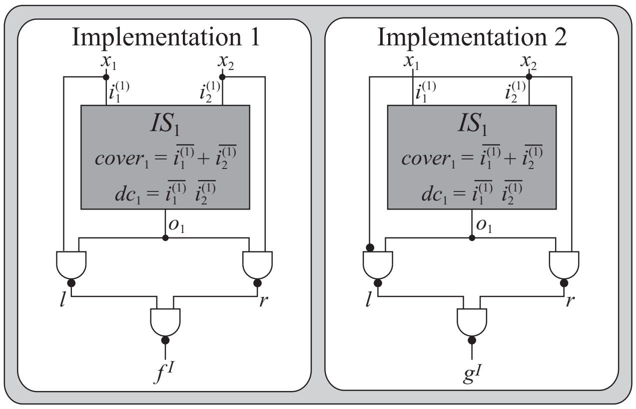
Incompletely Specified Boxes can occur, when a larger design is partitioned into blocks, where some blocks are incompletely specified. After a partition of the design into (completely or incompletely specified) blocks has been done in an early stage of the design process, it should be checked, whether the partitioning result is still equivalent to the specification. (This includes — among other things — the question whether the don't cares which were specified for IS–Boxes can really be used as don't cares in this implementation containing IS–Boxes.) The current implementation is only correct, if it is equivalent to the specification *for all* possible assignments to the don't cares of the IS–Boxes, since the designer of an IS–Box is allowed to assign the don't cares arbitrarily and the implementation has to be correct independently from the actual choice for the don't cares. An error is found, if *there is* an assignment to the don't cares of the IS–Boxes, such that implementation and specification differ for at least one primary input vector. We believe that incorrect assumptions about the behaviour of the environment of subcircuits are frequent sources of design errors.

Another possible application of IS–Boxes could be the use of *incompletely specified* Intellectual Property cores (IP cores) in an implementation. Under the assumption that the IP vendor does not publish the don't care assignment actually used in the design of the IP cores (for reasons of Intellectual Property protection), the IP cores have to be viewed as Incompletely Specified Boxes. The problem which has to be solved is to check whether specification and implementation with incompletely specified IP cores are equivalent. Again, specification and implementation with IS–Boxes can be called equivalent only if they are equivalent for all possible assignments to the don't cares of the IS–Boxes (incompletely specified IP cores).

Another application which does not fit into this framework at first sight can be handled by our solution, too: This problem occurs when *incomplete specifications* at the register transfer level are checked against *complete implementations* at the gate level. At the register transfer level it is natural to specify "unknowns" for cases when the value at the output of a block does not matter for the final result. This results in a specification with incompletely specified blocks, which is checked against a (complete) gate level implementation. Of course, the same techniques can be applied for the solution to this problem as in the two cases above; merely the roles of implementation and specification are exchanged. However, throughout the paper, we will always speak about complete specifications and incomplete implementations in order to avoid confusion.

In this paper we present a method to solve the problem of equivalence checking for implementations with IS–Boxes. The method is based on a transformation of the implementation into a circuit modeling the incompleteness and can be applied both in a BDD based and in a SAT based verification environment.

Figure 1. Two different implementations with an IS–Box. The specification of the whole circuit for both cases is $f^S = x_1 \oplus x_2$.



The paper is structured as follows: In Section 2 we give some preliminaries. The following section defines the problem of equivalence checking for implementations with Incompletely Specified Boxes, compares the notion of equivalence for implementations with IS–Boxes to the notion of equivalence for implementations with Black Boxes [14, 12, 26], and finally, it points out the relationship between our problem and the computation of satisfiability and observability don't cares. In Section 4 we present our solution to the problem. Our method is evaluated by experiments in Section 5. The paper ends with some concluding remarks and directions for further research in Section 6.

2. Preliminaries

Let $f : \{0,1\}^n \to \{0,1\}$ be a completely specified Boolean function with n inputs.

Incompletely specified Boolean functions with domain D are functions $f : D \to \{0,1\}$ with $D \subseteq \{0,1\}^n$. $\{0,1\}^n \setminus D$ is called 'don't care set' of f. An incompletely specified function with domain D can be represented by a pair $(cover, dc)$ of (completely specified) Boolean functions, where dc is a characteristic function of the don't care set, i.e., $dc : \{0,1\}^n \to \{0,1\}$ with $dc(\epsilon) = 1$ iff $\epsilon \notin D$ ($\forall \epsilon \in \{0,1\}^n$), and $cover : \{0,1\}^n \to \{0,1\}$ is an arbitrary function with the property $cover(\epsilon) = f(\epsilon)$ for all $\epsilon \in D$. A completely specified function $f' : \{0,1\}^n \to \{0,1\}$ is called a '(completely specified) extension' of $f : D \to \{0,1\}$ iff $f'(\epsilon) = f(\epsilon)$ for all $\epsilon \in D$.

Boolean functions can be represented by Reduced Ordered BDDs [1]. A BDD for a Boolean function can be computed by *symbolic simulation* of the corresponding circuit using the efficient manipulation algorithms for BDDs [2]. Then, due to the canonicity of BDDs the equivalence check for two circuit representations corresponds to a simple check for equality.

Another possibility to decide equivalence between two circuit representations of Boolean functions f and g is based on Boolean satisfiability. Starting from the circuit representations, the so-called miter circuit is constructed, which is satisfiable, if and only if f and g are not equivalent (see e.g. [19]). Then the miter circuit is transformed into a CNF formula using methods from [18, 19] and the equivalence check reduces to showing the unsatisfiability of the formula.

3. Incompletely Specified Boxes

Implementations containing *Incompletely Specified Boxes* (or *IS–Boxes*) are circuits which contain boxes whose outputs are associated with incompletely specified functions. Let o_i be an output of an IS–Box IS_j with input variables $i_1^{(j)}, \ldots, i_{l_j}^{(j)}$. Then o_i is associated with an incompletely specified function g_i, which may be represented by a pair of completely specified functions $(cover_i, dc_i)$, where $cover_i$ and dc_i depend on variables $i_1^{(j)}, \ldots, i_{l_j}^{(j)}$ and dc_i is the characteristic function of the don't care set of g_i.

Two small examples for implementations containing IS–Boxes are shown in Figure 1. The incompletely specified function g_1 associated with the IS–Box IS_1 in both examples is represented by $(cover_1, dc_1) = (\overline{i_1^{(1)}} + \overline{i_2^{(1)}}, \overline{i_1^{(1)}} \cdot \overline{i_2^{(1)}})$, i.e, the don't care set of g_1 is $\{(0,0)\}$. There are two possible completely specified extensions of g_1, namely the function $g_1'(i_1^{(1)}, i_2^{(1)}) = \overline{i_1^{(1)}} + \overline{i_2^{(1)}}$ (where the don't care $(0,0)$ is set to 1) and the function $g_1''(i_1^{(1)}, i_2^{(1)}) = i_1^{(1)} \oplus i_2^{(1)}$ (where the don't care $(0,0)$ is set to 0). The corresponding specification for the complete circuit is the function $f^S = x_1 \oplus x_2$.

3.1. Equivalence of specifications and implementations with IS–Boxes

When an IS–Box is used in a partition of a design into completely and incompletely specified boxes, implementation and specification can be called equivalent only if they represent the same Boolean function *for all possible replacements of the incompletely specified functions by completely specified extensions*, since the designer of an IS–Box has the freedom to choose an arbitrary assignment to the don't cares and the overall circuit has to be correct in any case.

In the example of Figure 1 this means that the implementation is only correct, if it fulfills its specification both for the replacement of the IS–Box IS_1 by function g_1' *and* for the replacement of IS_1 by g_1''.

In the next section we present a method to decide, whether specifications and implementations containing IS–Boxes are equivalent in the sense defined above.

3.2. Comparison to equivalence for implementations with Black Boxes

Before presenting an algorithm for equivalence checking we look into the relationship between the equivalence checking problem for implementations with IS–Boxes and the equivalence checking problem for implementations with Black Boxes [14, 12, 26].

At first sight the equivalence checking problem for implementations with Black Boxes seems to be a special case of the corresponding problem for IS–Boxes, since Black Boxes are boxes of which we do not know anything, i.e., they can be modeled by incompletely specified functions where the domain is empty. However, the two problems are applied in different contexts which leads to different notions of equivalence:

- As already mentioned, an implementation with IS–Boxes is *not correct*, if *there is* a replacement of the IS–Boxes by completely specified extensions, such that the resulting implementation does not fulfill its specification.
- In contrast, an implementation with Black Boxes is *not correct*, if *for all* replacements of the Black Boxes by some completely specified functions the resulting implementation does not fulfill its specification. A Black Box represents a part of the implementation which is not yet finished. Thus, an error can only be reported, if the partial implementation can not be completed to a correct overall implementation by any implementation for the Black Boxes.

So we have two completely different notions of equivalence for Black Boxes and IS–Boxes.

3.3. Relation to the computation of satisfiability and observability don't cares

The problem of equivalence checking for implementations with IS–Boxes is related to the computation of satisfiability and observability don't cares [9] used in logic synthesis[1]: In fact, we have to check, whether the don't cares given for the IS–Boxes are satisfiability and observability don't cares for the output functions of the boxes (under the assumption that the function of the overall circuit is defined by the specification). Thus, a naive approach to solve our problem — at least for the case that there is only one IS–Box with only one output — would be the computation of

[1] An input vector of some block in a circuit is a satisfiability don't care, if this input vector cannot be applied to the block due to the subcircuit computing the inputs of this block. An input vector of some block is an observability don't care for a certain output of the block, if the value at this output can be changed (for the input vector) without changing the overall behaviour of the circuit, i.e., if this change cannot be observed looking at the overall circuit.

the complete set of all satisfiability and observability don't cares for the IS–Box followed by a check, whether the given don't care set for the IS–Box is a subset of the computed set. However, the following facts prohibit this approach:

- For large circuits, the computation of satisfiability and observability don't cares can be expensive. For this reason, the computation of don't cares for a subcircuit usually considers only an environment of the subcircuit and not the whole circuit leading to a *subset* of the don't care set. However, we need the complete set of satisfiability and observability don't cares to prove that the given don't care set for an IS–Box is not correct.
- For IS–Boxes with several outputs and for several IS–Boxes in the implementation the situation will be much more complicated: In this case the flexibility has to be expressed by Boolean relations or by sets of Boolean relations, since the degree of freedom given at one output of an IS–Box depends upon the other outputs [30]. Also in this case we would need the *complete* information on flexibility for the IS–Boxes to decide whether the given don't care sets for the IS–Boxes are not correct.

Fortunately, we do not need to compute the complete information on flexibility for the IS–Boxes to solve our problem. In the following section we present a much simpler solution to the equivalence checking problem for implementations with IS–Boxes.

4. Solution to equivalence checking for implementations with IS-Boxes

In this section we present a method to solve the equivalence checking problem for implementations containing IS–Boxes. Our approach is based on the following idea: We use additional variables for the outputs of the IS–Boxes, which are used to model their function in the presence of don't care values at the inputs of IS–Boxes. The complete behaviour of the implementation can then be represented by a function depending on the circuit's primary input variables and the newly introduced variables. We obtain the following necessary and sufficient condition for equivalence: Specification and implementation are equivalent if and only if the resulting implementation function is exactly equal to the specification function. In particular, the implementation function finally must not depend on values assigned to the newly introduced variables.

We assume an implementation I, which has n primary inputs $x_1, \dots, x_n$ and b IS–Boxes $IS_1, \dots, IS_b$. The IS–Boxes have l ($l > b$) outputs $o_1, \dots, o_l$.[2] IS–Box IS_j depends on l_j input variables $i_1^{(j)}, \dots, i_{l_j}^{(j)}$. Each output o_i ($1 \leq i \leq l$) of an IS–Box IS_j is associated with an incompletely specified function represented by the pair $(cover_i, dc_i)$, where dc_i is the characteristic function of the don't care set for o_i. $cover_i$ and dc_i depend on the input variables $i_1^{(j)}, \dots, i_{l_j}^{(j)}$ of IS–Box IS_j.

[2] Each IS–Box has at least one output.

Now we introduce new variables $z_1, \dots, z_l$ for the outputs $o_1, \dots, o_l$ of the IS–Boxes. For each output o_i of an IS–Box IS_j we define a function

$$is_i(i_1^{(j)}, \dots, i_{l_j}^{(j)}, z_i) = \overline{dc_i} \cdot cover_i + dc_i \cdot z_i$$

depending on input variables $i_1^{(j)}, \dots, i_{l_j}^{(j)}, z_i$.

Let I_z be the circuit which results from the implementation with IS–Boxes by replacing the outputs of IS–Boxes by functions is_i, i.e., by replacing output o_i of IS–Box IS_j by function $is_i(i_1^{(j)}, \dots, i_{l_j}^{(j)}, z_i)$, respectively. Moreover, let $f_k^{I_z}(x_1, \dots, x_n, z_1, \dots, z_l)$ be the Boolean function computed by output k of I_z and let f_k^S be the corresponding output of the specification S.

Then the following theorem gives a necessary and sufficient condition for the correctness of the implementation with IS–Boxes:

Theorem 4.1 *Using the notations defined above the following two propositions are equivalent:*

1. *Output k of implementation I is not equivalent to output k of specification S, i.e., there is a replacement of the outputs o_i of IS–Boxes IS_j by completely specified extensions $cs_i(i_1^{(j)}, \dots, i_{l_j}^{(j)})$ leading to a circuit I_{cs}, such that $f_k^{I_{cs}}(x_1, \dots, x_n) \neq f_k^S(x_1, \dots, x_n)$.*
2. *$f_k^{I_z} \neq f_k^S$, i.e., there is $(\epsilon_1, \dots, \epsilon_n, \delta_1, \dots, \delta_l) \in \{0, 1\}^{n+l}$ with $f_k^{I_z}(\epsilon_1, \dots, \epsilon_n, \delta_1, \dots, \delta_l) \neq f_k^S(\epsilon_1, \dots, \epsilon_n)$.*

Proof: (sketch)

'2. $\Longrightarrow$ 1.':

Let $(\epsilon_1, \dots, \epsilon_n, \delta_1, \dots, \delta_l) \in \{0,1\}^{n+l}$ be an input vector which distinguishes between $f_k^{I_z}$ and f_k^S, i.e., $f_k^{I_z}(\epsilon_1, \dots, \epsilon_n, \delta_1, \dots, \delta_l) \neq f_k^S(\epsilon_1, \dots, \epsilon_n)$.

Then we define I_{cs} as follows: For each output o_i of an IS–Box IS_j we choose

$$cs_i(i_1^{(j)}, \dots, i_{l_j}^{(j)}) = \overline{dc_i} \cdot cover_i + dc_i \cdot \delta_i,$$

i.e., all don't cares of output o_i of IS–Box IS_j are fixed to δ_i.

Under the assumption that input vector $(\epsilon_1, \dots, \epsilon_n)$ is applied to circuit I_{cs} and input vector $(\epsilon_1, \dots, \epsilon_n, \delta_1, \dots, \delta_l)$ is applied to circuit I_z, we can prove that all corresponding signals in I_{cs} and I_z [3] have the

[3] Since both I_{cs} and I_z result from I by replacement of IS–Boxes, 'corresponding signals' are signals which result from identical signals in I.

same value. The proof is done by induction on the depth of the signals. It is clear that corresponding primary inputs x_i (i.e. signals with depth 0) have the same values ϵ_i. The interesting case for signals with depth $d > 0$ occurs when a signal is computed by an output o_i of an IS–Box IS_j. The input signals $i_1^{(j)}, \dots, i_{l_j}^{(j)}$ of IS_j are corresponding signals in I_{cs} and I_z and — according to the induction hypothesis — have the same values $(\alpha_1, \dots, \alpha_{l_j})$. Now the functions $cs_i = \overline{dc_i} \cdot cover_i + dc_i \cdot \delta_i$ and $is_i = \overline{dc_i} \cdot cover_i + dc_i \cdot z_i$ for o_i in I_{cs} and I_z, respectively, have the same values: If $dc_i(\alpha_1, \dots, \alpha_{l_j}) = 0$, then the identical value is $cover_i(\alpha_1, \dots, \alpha_{l_j})$ and if $dc_i(\alpha_1, \dots, \alpha_{l_j}) = 1$, then the identical value is δ_i.

Since all corresponding signals in I_{cs} and I_z have the same value, we can conclude that $f_k^{I_{cs}}(\epsilon_1, \dots, \epsilon_n) = f_k^{I_z}(\epsilon_1, \dots, \epsilon_n, \delta_1, \dots, \delta_l)$ and thus $f_k^{I_{cs}}(\epsilon_1, \dots, \epsilon_n) \neq f_k^S(\epsilon_1, \dots, \epsilon_n)$.

'1. $\Longrightarrow$ 2.':

Assume that there is a replacement of the outputs o_i of IS–Boxes IS_j by completely specified extensions $cs_i(i_1^{(j)}, \dots, i_{l_j}^{(j)})$, such that $f_k^{I_{cs}}(\epsilon_1, \dots, \epsilon_n) \neq f_k^S(\epsilon_1, \dots, \epsilon_n)$ for $(\epsilon_1, \dots, \epsilon_n) \in \{0,1\}^n$.

The outputs o_i of IS–Boxes IS_j are associated with incompletely specified functions represented by $(cover_i, dc_i)$ $(1 \leq i \leq l)$. Since cs_i is a completely specified extension of the incompletely specified function represented by $(cover_i, dc_i)$, we have $cs_i = \overline{dc_i} \cdot cover_i + dc_i \cdot h_i$ for some Boolean function h_i depending on variables $i_1^{(j)}, \dots, i_{l_j}^{(j)}$.

For primary input vector $(\epsilon_1, \dots, \epsilon_n)$ let δ_i be the values computed by outputs o_i of IS–Boxes IS_j in circuit I_{cs} $(1 \leq i \leq l)$. Now we claim that $f_k^{I_z}(\epsilon_1, \dots, \epsilon_n, \delta_1, \dots, \delta_l) = f_k^{I_{cs}}(\epsilon_1, \dots, \epsilon_n)$.

As in part '2. $\Longrightarrow$ 1.', we prove by induction on the depth of signals that input vectors $(\epsilon_1, \dots, \epsilon_n)$ and $(\epsilon_1, \dots, \epsilon_n, \delta_1, \dots, \delta_l)$ lead to the same values for corresponding signals in I_{cs} and I_z. It is easy to see that this is true for signals with depth 0 and for signals which are computed by gates different from outputs of IS–Boxes. Now consider a signal which is computed by output o_i of an IS–Box IS_j. Let $(\alpha_1, \dots, \alpha_{l_j})$ be the vector of values for the input signals $i_1^{(j)}, \dots, i_{l_j}^{(j)}$ of IS_j in I_{cs} and I_z. If $dc_i(\alpha_1, \dots, \alpha_{l_j}) = 0$, then both is_i in I_z and cs_i in I_{cs} compute $cover_i(\alpha_1, \dots, \alpha_{l_j})$. If $dc_i(\alpha_1, \dots, \alpha_{l_j}) = 1$, then cs_i computes $h_i(\alpha_1, \dots, \alpha_{l_j}) = \delta_i$ and $is_i(i_1^{(j)}, \dots, i_{l_j}^{(j)}, z_i) = \overline{dc_i} \cdot cover_i + dc_i \cdot z_i$ computes δ_i, too, since z_i is set to δ_i by the assignment made to the primary inputs of I_z.

□

Based on Theorem 4.1 we can use the following BDD based method to check equivalence of specifications and implementation containing IS–Boxes:

1. Replace each output o_i of an IS–Box IS_j, which is associated with an incompletely specified function represented by $(cover_i, dc_i)$, by the function $is_i = \overline{dc_i} \cdot cover_i + dc_i \cdot z_i$ with a new variable z_i.
2. Compute for each output k of the resulting circuit I_z the BDD for $f_k^{I_z}$ by symbolic simulation.
3. Compute for each output k of the specification the BDD for f_k^S by symbolic simulation.
4. Check whether the BDDs for $f_k^{I_z}$ and f_k^S are different for some primary output k. If this is the case, then specification and implementation are not equivalent, otherwise they are equivalent.

As an alternative, we can also perform a SAT based equivalence checking: We add a miter circuit to the circuits S and I_z after step 1. of the algorithm above (i.e., connect each output k of S and each output k of I_z to an *exor* gate and connect all *exor* gates to an *or* gate) and then we check whether the output of the resulting circuit is satisfiable.

Example 4.1 Consider the implementations in Figure 1. Both implementations contain one IS–Box realizing an incompletely specified function represented by $(cover_1, dc_1) = (\overline{i_1^{(1)}} + \overline{i_2^{(1)}}, \overline{i_1^{(1)}} \cdot \overline{i_2^{(1)}})$. The corresponding specification is the function $f^S = x_1 \oplus x_2$.

For symbolic simulation, the IS–Box is replaced by $is_1 = \overline{dc_1} \cdot cover_1 + dc_1 \cdot z_1 = (i_1^{(1)} + i_2^{(1)})(\overline{i_1^{(1)}} + \overline{i_2^{(1)}}) + \overline{i_1^{(1)}}\ \overline{i_2^{(1)}}\ z_1$. For each gate, symbolic simulation computes the function realized by the gate depending on primary inputs. Gates are processed in topological order.

For Implementation 1, symbolic simulation computes

- for o_1: $f_{o_1}^{I_z} = (x_1 + x_2)(\overline{x_1} + \overline{x_2}) + \overline{x_1}\ \overline{x_2}\ z_1 = x_1\overline{x_2} + \overline{x_1}x_2 + \overline{x_1}\ \overline{x_2}\ z_1$,
- for l: $f_l^{I_z} = \overline{x_1 \cdot f_{o_1}^{I_z}} = \overline{x_1 \cdot \overline{x_2}}$,
- for r: $f_r^{I_z} = \overline{x_2 \cdot f_{o_1}^{I_z}} = \overline{\overline{x_1} \cdot x_2}$,
- and finally $f^{I_z} = \overline{f_l^{I_z}} + \overline{f_r^{I_z}} = x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2 = x_1 \oplus x_2$.

Since $f^{I_z} = f^S$, the implementation is equal to the specification *independently from the assignment to the don't cares of the IS–Box* and thus implementation and specification are equivalent.

For Implementation 2, symbolic simulation computes

- for o_1: $g_{o_1}^{I_z} = (x_1 + x_2)(\overline{x_1} + \overline{x_2}) + \overline{x_1}\ \overline{x_2}\ z_1 = x_1\overline{x_2} + \overline{x_1}x_2 + \overline{x_1}\ \overline{x_2}\ z_1$,

Table 1. 10% of the gates in one IS–Box

			Specification		Implementation, 10%, 1 Box				
						no error		error	
circuit	in	out	#nodes	time	%dc	#nodes	time	#nodes	time
alu4	14	8	530	0.61	25%	378	0.24	391	0.10
apex7	49	37	294	0.14	10%	295	0.06	269	0.06
comp	32	3	137	0.08	37%	137	0.05	117	0.04
term1	34	10	82	0.12	35%	81	0.06	88	0.06
C432	36	7	1212	0.27	58%	1240	0.20	1699	0.22
C499	41	32	30104	11.45	60%	37148	8.77	38770	9.43
C880	60	26	4814	1.83	27%	4662	0.50	4674	0.54
C1355	41	32	26901	11.06	32%	30581	4.02	35529	7.56
C1908	33	25	7161	4.70	54%	7040	1.63	7165	1.85
C2670	233	140	2141	5.21	22%	3784	1.62	4374	1.97
C3540	50	22	36114	20.43	19%	36855	4.89	37566	5.82
C5315	178	123	1989	3.66	5%	1932	1.21	1954	1.22
C7552	207	108	8545	17.44	20%	10908	4.67	8330	3.99

Table 2. 10% of the gates in five IS–Boxes

			Specification		Implementation, 10%, 5 Boxes				
						no error		error	
circuit	in	out	#nodes	time	%dc	#nodes	time	#nodes	time
alu4	14	8	530	0.63	99%	447	0.52	489	0.23
apex7	49	37	256	0.15	42%	273	0.07	249	0.07
comp	32	3	137	0.08	71%	137	0.04	138	0.04
term1	34	10	91	0.14	45%	81	0.06	92	0.07
C432	36	7	1212	0.25	23%	1296	0.18	1515	0.15
C499	41	32	26408	10.87	50%	25974	2.70	33758	5.37
C880	60	26	4738	1.91	22%	4938	0.52	4750	0.56
C1355	41	32	26085	10.38	91%	27425	3.52	36805	7.71
C1908	33	25	7108	4.77	99%	7661	3.78	7391	3.60
C2670	233	140	2377	5.77	91%	2193	1.38	4352	2.09
C3540	50	22	36154	30.39	85%	30160	9.52	30271	7.57
C5315	178	123	2199	3.61	69%	2386	1.39	2224	1.34
C7552	207	108	22751	46.38	79%	15724	7.80	13720	7.09

- for l: $g_l^{I_z} = \overline{\overline{x_2} \cdot g_{o_1}^{I_z}} = \overline{x_1 \cdot \overline{x_2} + \overline{x_1}\,\overline{x_2}\,z_1}$,
- for r: $g_r^{I_z} = \overline{x_2 \cdot g_{o_1}^{I_z}} = \overline{\overline{x_1} \cdot x_2}$,
- and finally $g^{I_z} = \overline{g_l^{I_z}} + \overline{g_r^{I_z}} = x_1 \cdot \overline{x_2} + \overline{x_1}\,\overline{x_2}\,z_1 + \overline{x_1} \cdot x_2$.

Since $g^{I_z} \neq f^S = x_1 \oplus x_2$, implementation and specification are *not* equivalent. Note that in this example there is a don't care assignment for the IS–Box, such that implementation and specification will become equal. If the output of the IS–Box for input $(0,0)$ is chosen to be 0, then implementation and specification will be equal. However, if the output of the IS–Box for input $(0,0)$ is 1, then implementation and specification will *not* be equal and thus, the specification and the implementation containing an IS–Box are called *not equivalent*, since the implementation has to fulfill its specification for all possible assignments to the don't cares.

5. Experimental results

To evaluate our method for equivalence checking in the presence of IS–Boxes we implemented the basic procedures

Table 3. 40% of the gates in one IS–Box

			Specification		Implementation, 40%, 1 Box				
						no error		error	
circuit	in	out	#nodes	time	%dc	#nodes	time	#nodes	time
alu4	14	8	384	0.21	16%	361	0.05	383	0.06
apex7	49	37	294	0.14	10%	295	0.06	269	0.06
comp	32	3	137	0.07	29%	147	0.04	125	0.04
term1	34	10	82	0.13	30%	81	0.06	82	0.06
C432	36	7	1321	0.54	44%	2119	1.66	2046	0.58
C499	41	32	30104	11.34	60%	41640	21.13	33979	14.65
C880	60	26	4814	1.85	27%	4662	0.50	4674	0.54
C1355	41	32	26901	11.07	32%	28986	16.35	35573	17.09
C1908	33	25	8371	35.32	60%	7872	60.88	7534	23.24
C2670	233	140	2141	5.26	2%	3004	1.40	4118	1.93
C3540	50	22	36114	20.50	19%	36855	4.91	37566	5.83
C5315	178	123	1989	3.63	5%	1932	1.21	1955	1.22
C7552	207	108	10466	24.39	9%	9738	3.88	8348	3.73

as described in the previous sections using a BDD-based and a SAT-based approach, as well.

Although, for the purpose of this paper, our implementation is restricted to these two extreme cases, it should be mentioned that it is straightforward to incorporate other approaches combining the strengths of BDD based and SAT based methods by a tight integration [13, 6, 23, 16]. Furthermore, for specifications and implementations having structural similarities, methods which identify equivalences between internal nodes to simplify the verification problem [17, 6, 23] are applicable to our problem, too.

For our implementation we used *CUDD 2.3.0* [28] as the underlying BDD package. Dynamic reordering [25] was activated during all BDD-based experiments. Furthermore, we used *Chaff* [22] as satisfiability solver in a series of experiments based on a satisfiability formulation of the problem. The experiments were performed on a PentiumIII PC running Linux (550 MHz, 1 GB memory).

To obtain circuits with IS–Boxes we generated implementations with IS–Boxes from benchmark circuits: For each benchmark circuit a certain fraction of the gates was randomly selected to form IS–Boxes. In a first experiment (Table 1) we included 10% of the gates in one IS–Box, in a second experiment (Table 2) 10% of the gates in five IS–Boxes and in a third experiment (Table 3) 40% of the gates in one IS–Box. The incompletely specified functions for the outputs of the IS–Boxes were computed based on the original benchmark circuit: For an output o_i of an IS–Box the incompletely specified function is represented by a pair $(cover_i, dc_i)$. The function $cover_i$ was chosen as the function originally implemented in the benchmark circuit and the function dc_i was obtained by computing satisfiability and observability don't cares [9] for the IS–Box. Both $cover_i$ and dc_i were represented by a BDD and were associated with the IS–Box.

The original benchmark circuit was used as the specification.

In a first set of experiments we used the circuit with IS–

Boxes computed as described above as the implementation. In this case, specification and implementation are equivalent, since the don't cares for the outputs of the IS–Boxes are satisfiability or observability don't cares of the Boxes in the original benchmark circuit. In a second set of experiments we inserted errors into the implementations with IS–Boxes to check the robustness of our method for erroneous implementations. We randomly selected a gate, which did not belong to an IS–Box, and inserted an error. 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 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 reported results are an average on 5 different random selections of IS–Boxes and in the case of error insertions also an average on 100 different random selections of error insertions.[4]

Tables 1, 2 and 3 show the results of BDD-based experiments. In columns 1, 2, and 3 the names of the benchmarks, and the numbers of inputs and outputs are given. Column 4 gives the numbers of ROBDD nodes needed to represent the specification; column 5 shows the CPU times in seconds to compute the BDD representation of the specification. Column 6 gives the average fraction of minterms, which are used as don't cares for the different outputs of the IS–Boxes. Then the results for two sets of experiments are presented: For the first experiment, where no errors were inserted, columns 7 and 8 show the numbers of BDD nodes and the CPU times needed for symbolic simulation of the implementation and in columns 9 and 10 the same information is given for the second experiment using error insertions.

The observations made based on the experiments are the same for Tables 1, 2 and 3 (10% of the gates included in one IS–Box, 10% of the gates included in five IS–Boxes and 40% of the gates included in one IS–Box): The symbolic simulation of implementations with IS–Boxes shows no performance degradation at all compared with the symbolic simulation of the specification. This observation is true for different sizes and numbers of IS–Boxes. In our experiments this property also did not change when errors were inserted into the implementations with IS–Boxes. This clearly demonstrates the feasibility of our approach.

We also evaluated the method in connection with equivalence checking based on Boolean satisfiability. The problem was translated into a satisfiability problem as described in Section 4 and then it was transformed into a CNF formula using methods from [18, 19]. Table 4 shows the results for the experiment, where 10% of the gates were included in one IS–Box. Columns 4 and 5 repeat the results for the BDD based approach[5]. For the instances where no errors were inserted columns 6, 7, and 8 show the sizes of the corresponding SAT problems in terms of variables, clauses, and literals, respectively; column 9 gives the corresponding CPU times in seconds. Columns 10–13 give the same information for the instances where errors were inserted. The results show that the SAT based approach can compete with the BDD based approach for the problems with errors (where the corresponding CNF formula is satisfiable). The instances without errors which were solved by the BDD based approach could also be solved by the SAT based approach, but only with (sometimes considerably) larger CPU times. For circuit $C6288$ (which is a 16x16 multiplier) the individual strengths of the SAT based approach become apparent: With a memory limit of 1 GB and a time limit of 100.000 CPU seconds, neither the correct nor the erroneous instances could be solved using BDDs. Whereas the correct instances could not be solved by the SAT engine either (using the same memory and time limits), the erroneous instances could be solved within a few seconds.

[4] Of course, the circuit modifications described above do not necessarily lead to errors in the implementation. For our results we used only those modifications, which really lead to errors.

6. Conclusion and future work

In this paper we introduced the concept of equivalence checking for implementations with IS–Boxes. The relationship of this notion of equivalence to the equivalence for implementations containing Black Boxes was clarified. We presented a method to solve the problem of equivalence checking for implementations with IS–Boxes using a transformation of the implementation.

Experimental results using both, a BDD based and a SAT based approach, prove the feasibility of the approach. Since after the transformation of Section 4 the remaining task consists in comparing two combinational circuits, an incorporation of recently presented methods combining the strengths of BDD based and SAT based algorithms by a tight integration is straightforward. For the same reason also methods to identify equivalences between internal nodes to simplify the verification problem for circuits with structural similarities can be applied to our approach.

For the future, we are planning to generalize the approach to the verification of sequential circuits containing IS–Boxes.

References

[1] R. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[2] R. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM, Comp. Surveys*, 24:293–318, 1992.

[5] The CPU times are the sums of the times for computing the BDDs of specification and implementation.

Table 4. 10% of the gates in one IS–Box, BDDs versus SAT

			BDD		SAT							
			no error	error	no error				error			
circuit	in	out	time	time	Var.	Clauses	Lit.	time	Var.	Clauses	Lit.	time
alu4	14	8	0.85	0.71	1726	7488	20817	29.78	1726	7488	20816	3.52
apex7	49	37	0.20	0.20	460	1539	3988	1.64	460	1539	3987	0.28
comp	32	3	0.13	0.12	280	833	2063	0.49	280	833	2062	0.18
term1	34	10	0.18	0.18	1053	3906	9881	17.90	1053	3906	9881	4.27
C432	36	7	0.47	0.49	956	3425	9663	18.88	956	3425	9663	0.92
C499	41	32	20.22	20.88	3237	12123	34976	1923.12	3237	12122	34974	170.34
C880	60	26	2.33	2.37	858	2653	6822	55.60	857	2653	6821	1.33
C1355	41	32	15.08	18.62	2560	9203	25769	3783.42	2560	9203	25768	22.81
C1908	33	25	6.33	6.55	5804	22360	65339	9438.22	5804	22360	65339	51.96
C2670	233	140	6.83	7.18	2597	8162	21746	126.09	2597	8162	21745	6.76
C3540	50	22	25.32	26.25	2565	8661	22191	894.97	2564	8661	22190	93.85
C5315	178	123	4.87	4.88	3187	10518	26069	732.98	3187	10518	26069	28.62
C6288	32	32	-	-	-	-	-	-	4921	14680	34571	49.39
C7552	207	108	22.11	21.43	8480	29635	80553	3605.43	8480	29635	80553	90.64

[3] R. Bryant. Bit-level analysis of an SRT divider circuit. In *Design Automation Conf.*, pages 661–665, 1996.

[4] R. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Design Automation Conf.*, pages 535–541, 1995.

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

[6] J. Burch and V. Singhal. Tight integration of combinational verification methods. In *Int'l Conf. on CAD*, pages 570–576, 1998.

[7] E. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams - overcoming the limitations of MTBDDs and BMDs. In *Int'l Conf. on CAD*, pages 159–163, 1995.

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

[9] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[10] R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs: A new data structure for verification. In *European Design & Test Conf.*, pages 2–8, 1996.

[11] E. Goldberg, M. Prasad, and R. Brayton. Using SAT for combinational equivalence checking. In *Int'l Workshop on Logic Synth.*, pages 185–191, 2000.

[12] W. Günther, N. Drechsler, R. Drechsler, and B. Becker. Verification of designs containing black boxes. In *EUROMICRO*, pages 100–105, 2000.

[13] A. Gupta and P. Ashar. Integrating a boolean satisfiability checker and BDDs for combinational equivalence checking. In *VLSI Design*, 1998.

[14] 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.

[15] J. Jain, R. Mukherjee, and M. Fujita. Advanced verification techniques based on learning. In *Design Automation Conf.*, pages 420–426, 1995.

[16] A. Kuehlmann, M. Ganai, and V. Paruthi. Circuit-based Boolean reasoning. In *Design Automation Conf.*, pages 232–237, 2001.

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

[18] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Trans. on CAD*, 11:4–15, 1992.

[19] J. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Design, Automation and Test in Europe*, pages 145–149, 1999.

[20] Y. Matsunaga. An efficient equivalence checker for combinational circuits. In *Design Automation Conf.*, pages 629–634, 1996.

[21] 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.

[22] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engeneering an efficient SAT solver. In *Design Automation Conf.*, 2001.

[23] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. In *Int'l Conf. on Comp. Design*, pages 459–464, 2000.

[24] S. Reddy, W. Kunz, and K. Pradhan. Novel verification framework combining structural and OBDD methods in a synthesis environment. In *Design Automation Conf.*, pages 414–419, 1995.

[25] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.

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

[27] H. Sharangpani and M. Barton. Statistical analysis of floating point flaw in the pentium processor (1994). Technical report, Intel Technical Report, Nov. 30, 1994.

[28] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.0*. University of Colorado at Boulder, 1998.

[29] P. Tafertshofer, A. Ganz, and M. Henftling. A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists. In *Int'l Conf. on CAD*, pages 648 – 655, 1997.

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