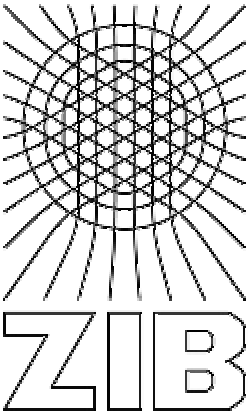
Konrad-Zuse-Zentrum
für Informationstechnik Berlin



Takustraße 7
D-14195 Berlin-Dahlem
Germany

Yakov Novikov Raik Brinkmann

Foundations of Hierarchical SAT-Solving

ZIB-Report 05-38 (August 2005)

6th International Workshop on "Boolean Problems" 2004, Freiberg University of Mining and Technology, Institute of Computer Science, September 23-24, pp. 103-142, 2004

Foundations of Hierarchical SAT-Solving*)

Yakov Novikov
Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
yakov_nov@tut.by

Raik Brinkmann
Infineon Technologies AG München
Raik.Brinkmann@infineon.com

Abstract

The theory of hierarchical Boolean satisfiability (SAT) solving proposed in this paper is based on a strict axiomatic system and introduces a new important notion of implicativity. The theory makes evident that increasing implicativity is the core of SAT-solving. We provide a theoretical basis for increasing the implicativity of a given SAT instance and for organizing SAT-solving in a hierarchical way. The theory opens a new domain of research: SAT-model construction. Now quite different mathematical models can be used within practical SAT-solvers. The theory covers many advanced techniques such as circuit-oriented SAT-solving, mixed BDD/CNF SAT-solving, merging gates, using pseudo-Boolean constraints, using state machines for representation of Boolean functions, arithmetic reasoning, and managing don't cares. We believe that hierarchical SAT-solving is a cardinal direction of research in practical SAT-solving.

Keywords: Hierarchical SAT-solving, implicativity

1 Introduction

Currently, state-of-the-art SAT-solvers are becoming the main workhorse in formal verification [1-6]. It is intuitively clear that further progress in SAT-solving may be achieved by increasing the level of abstraction used by SAT-solvers, as they should greatly benefit from the high level information. For combinatorial equivalence checking it has been shown [7] that knowing the high level structure of a system under consideration is of great theoretical and practical importance. In this paper, we provide theoretical foundations of hierarchical SAT-solving.

Many tasks from the verification domain, as well as from others, boil down to the question whether a discrete function with 0-1 domain and finite co-domain is constant. Given a representation of such a function this problem can be formulated as a Boolean SAT problem by constructing a suitable combinatorial circuit $\boldsymbol{G}$ and verifying whether a given output realizes a given constant Boolean function (0 or 1). For example, various approaches for SAT-model construction lead to this formulation both in hardware verification [1-5], and software verification [6]. Usually, the gates of a circuit $\boldsymbol{G}$ implement either elementary Boolean functions (low level), bit vector arithmetic or logic functions (such as addition, comparison, shifting, and multiplexing) on the high level, as opposed to arbitrary Boolean functions.

Currently, practical SAT-solving (e.g. for Bounded-Model-Checking) is done on the lowest possible level of abstraction. A problem description that is given on the high level is transformed into low (gate) level and again into conjunctive normal form (CNF). This procedure has two main drawbacks: 1) important information about circuit organization on the high level is lost; 2) the size of the problem under consideration is (dramatically) increased.

Our theory is formulated to overcome these drawbacks. We view gates as blocks and a combinatorial circuit as a system of blocks to emphasize the fact that blocks can implement complex functions. (We adopt the concept of blocks for subsequent considerations because it provides an intuitive way of understanding. However, our approach is not restricted to the hardware domain.) The basic idea is to operate on blocks during SAT-solving similar to the way clauses are handled in CNFs currently. Since clauses are used to fix conflicts and to deduce implications, the same ability should be incorporated into block models.

*) This is a corrected and extended version of the paper. Basic changes are described in Appendix 3.

Our proposed technique is based on two main principles, "*encapsulation*" and "*divide and conquer*". Encapsulation means that for typical blocks or structures of the application domain special models are used during SAT-solving which encapsulate information (related to fixing conflicts and deducing implications). Conceptually, a model can be viewed as a container storing relevant information in a compact form and allowing for quick access. It is possible that much effort is required to develop good models for standard blocks, however, these models can be reused effectively in SAT-solving.

The principle divide and conquer complements the first one. When the workload is distributed among block models, the SAT-solver effectively becomes a coordinator. The evident gain is that a SAT-solver needs to branch only on variables describing interconnections between blocks. This results in a reduction of the search space in comparison to branching on all internal block variables.

Note that the search space can be greatly reduced by branching only on the inputs of a combinatorial circuit. In this case, SAT-solving is in fact reduced to simulation using Boolean constraint propagation. However, modern practical SAT-algorithms try to encounter conflicts as early as possible (e.g. by using special decision making strategies such as VSIDS [8] or similar [9,10], as well as signal correlation [11]) and deduce new conflict clauses based on clauses topologically close to the conflicting variables (first UIP conflict analysis strategy [12,13]). Our theory allows using these popular techniques. In addition it provides a chance to increase the performance of SAT-tools based on the principles of "encapsulation" and "divide and conquer".

Both principles potentially reduce the overall SAT-solving time and validate the concept of hierarchical SAT-solving. We call our approach hierarchical SAT-solving because it allows for system structure and uses models of blocks which potentially can be of any complexity. A block model can be a system of smaller blocks, as well as any mathematical construction satisfying our proposed axiomatic system.

The efficiency of the SAT-solving process highly depends on the properties of the mathematical models used in each block. For example, suppose that a block model can propagate signals only forward (from inputs to outputs). Then this is restrictive for a SAT-solver. For instance, two outputs of a block may be assigned to 1 whereas this combination of signals is inconsistent with respect to implemented function. In this case, a contradiction is not detected until all possible value combinations for block inputs have been tried. In the rest of the paper, we refer to inputs and outputs of a block as pins.

Now the following questions arise: a) What properties should models of system blocks possess to be most beneficial from the point of view of hierarchical SAT-solving? b) How can a given model be improved, if it is not good enough? c) How should the hierarchical SAT-solving process be organized? The key notion introduced here to answer these questions is "*implicativity*". Implicativity is a number conceptually characterizing the ability of a block to propagate values between its pins. We show that, if a system (considered as a big block) has maximal implicativity, then SAT-solving for the system becomes trivial. We show how the implicativity of a model can be measured, increased and extended up to maximal possible.

The theory proposed here shows that hierarchical SAT-solving boils down to increasing the implicativity of a system. In particular, this can be done by increasing the implicativity of its blocks (e.g. during preprocessing). In general, developing models with maximal implicativity is a challenge, as they have to be able to encapsulate an exponential number of conflicts and implicates. Our optimism is based on the observation that often typical and regular structures are used for function representation. Therefore, SAT model designers may focus on these particular cases. We show that traditional models, including but not limited to CNFs and binary decision diagrams (BDDs), can be used as block models within our framework. It is important to underline that a mathematical model of a block can be based on a more powerful system of calculus than general resolution. As a result, hierarchical SAT-solving can be more powerful in this context than general resolution.

In this work, we show how recent directions of research, currently disjoint but proven to be efficient, fit into our framework. The considered examples are circuit-oriented SAT-solving [11,14,15], mixed BDD/CNF SAT-solving [16-20], merging gates [4,21-23], using pseudo-Boolean constraints [24], using state machines for representation of Boolean functions [25], arithmetic reasoning [26], and managing don't cares [27]. All theses techniques directly or implicitly construct block models with maximal or increased implicativity. Our theory opens a rich domain for incorporating other mathematical models into practical SAT-solvers.

The paper is organized as follows. First, we introduce our axiomatic system (Section 3) and present a detailed example (Section 3). Then we formally introduce the concept of implicativity and show how it can be measured (Sections 4 and 5). The fundamental theorem on hierarchical SAT-solving, which

states that the SAT-solving problem for a block with maximal implicativity is trivial, is presented in Section 6. Consistent models are discussed in Section 7. Section 8 shows how the implicativity of a complex block can be estimated. In Section 9, we show that a system of blocks can be treated as any other block. On this basis, hierarchical SAT solving can be implemented by using different kinds of blocks (Section 10). In Section 11, we show how other techniques fit into our framework and how they can be extended. Finally, we conclude and present some ideas for extending this work.

2 Model Specification

We consider blocks on two levels of abstraction. On the low level, the internal organization of a block is of great importance. There, the main interest is the mathematical model used within the block, and how it is implemented. We presume that this mathematical model can be arbitrary, for example, a formula, an algorithm, a combinatorial circuit, or any other system of calculus.

On the high level, the internal structure of a block and the nature of its mathematical model do not matter (encapsulation). We focus on its communicative aspects only, i.e. we consider a blocks response under a partial value assignment to its pin variables. Conceptually, we distinguish three types of block responses. First, a block may recognize that such an assignment is inconsistent with respect to its mathematical model. In this case, it must report that this assignment is conflicting. Second, if the affecting assignment is not classified as conflicting, the block can produce a value assignment to some other pin variables. This action can be interpreted on the high level as proving an implication, i.e. the assignment implies the response. Third, the block can provide no new information under the assignment, i.e. the assignment is recognized neither as conflicting nor implying.

It is easy to see that the information provided by blocks in our framework suffices to solve SAT-problems on higher levels of abstraction by branching only on their pin variables. Our goal is to define "good behavior" in this context in a strict axiomatic way. It provides the basis for developing good low level SAT-models to facilitate efficient hierarchical SAT-solving. Furthermore, it can help to reveal bottlenecks in a system under consideration, i.e. "bad regions" that should be treated very carefully during SAT-solving, leading to "clever" heuristics for SAT-solvers.

Now we introduce our system of axioms which each block has to satisfy. The fist two axioms provide the necessary conditions for a block to behave like a usual combinatorial gate.

***Axiom* 1**. A block $\boldsymbol{B}$ has a finite nonempty set of Boolean input and output variables, called *pin variables* of the block. ⊗

***Axiom* 2**. A block $\boldsymbol{B}$ corresponds to a Boolean vector function $\boldsymbol{y} = Y(\boldsymbol{x})$, where $\boldsymbol{x}$ is a vector of input variables of the block, and $\boldsymbol{y}$ is a vector of output variables of the block. ⊗

Note, if a block has no inputs, a constant Boolean function is implemented on each its output. If a block $\boldsymbol{B}$ has no outputs and $\boldsymbol{x}$ is the vector of its input variables, then this block corresponds to the "idle" Boolean function $\varnothing = Y(\boldsymbol{x})$ which is an abstract mathematical construction used for completeness of subsequent consideration.

A variable value assignment a will refer to a set of equalities of type $x = d$ where $d \in \{0, 1\}$. An equality $x = d$ means that variable x must be assigned to d and is called an elementary assignment. We consider only assignments a in which no variable is assigned to opposite values simultaneously. Given a set of variables S, a *partial assignment* a to variables of S will be referred to as a value assignment to a subset S' of S, i.e. $S' \subseteq S$. If $S' = S$, we also call a a *full* (or *complete*) *assignment* to S, and if $S' = \varnothing$, a is an *empty assignment* to S.

***Axiom* 3**. A block $\boldsymbol{B}$ has a mathematical model $\boldsymbol{M}$ for which a procedure "*Boolean constraint propagation"* (BCP) is defined on the set of partial value assignments to its pin variables as follows. Given a partial assignment a,

a) BCP does not change any elementary assignment to a variable present in a.
b) BCP classifies whether the assignment a belongs to one of the possible abstract types *conflicting* or *not conflicting*.
c) If the assignment a is not conflicting, BCP can assign values to some unassigned pin variables (opposite values can not be assigned to the same variable simultaneously), which is denoted as b = BCP(a) where b is an additional assignment and $g = a \cup b$ is a resulting assignment to pin variables after running BCP**.** ⊗

If an assignment a is not conflicting and $b \neq \varnothing$, where $b = \mathrm{BCP}(a)$, then a *implies* the assignment b and we call a an *implying* assignment. In this case, the model $\boldsymbol{M}$ is said to *produce* the implication $a \Rightarrow b$ (or $a \Rightarrow b$ for $\boldsymbol{M}$). Let b be an elementary assignment (i.e. a value assignment to only one variable), then we call $a \Rightarrow b$ *elementary implication*. Now if b_i is an elementary assignment such that $b_i \in b$, then $a \Rightarrow b_i$ is an *elementary implication* of the implication $a \Rightarrow b$ (or $a \Rightarrow b_i$ is an elementary implication for $\boldsymbol{M}$).

Thus, the behavior of a block is determined by its BCP-procedure. We use the notion of procedure to emphasize the algorithmic nature of the model $\boldsymbol{M}$. The model stores information which is extracted by the BCP-procedure. We use Boolean constraint propagation for bridging the gap with respect to traditional BCP on CNF, moreover, as we show in Section 3, the traditional BCP for CNFs satisfies our system of axioms. In general, BCP-procedures can be quite different depending on the models used.

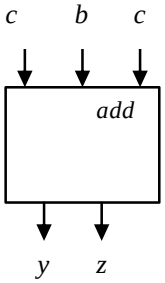
Now, we introduce an axiom on monotony which is naturally satisfied by realistic models as shown in the sequel.

***Axiom* 4**. (*Monotone classification*) Let a be a partial assignment to the pin variables of a block $\boldsymbol{B}$, and let $\boldsymbol{M}$ be a model of $\boldsymbol{B}$.

a) If an assignment a is conflicting for $\boldsymbol{M}$, then any assignment $a\phi$ containing a is conflicting for $\boldsymbol{M}$.
b) For any elementary implication $a \Rightarrow b_i$ for $\boldsymbol{M}$ and for any assignment g such that $g \cap (a \cup b_i \cup \neg b_i) = \varnothing$, there is an elementary implication $a \cup g \Rightarrow b_i$ or $a \cup g$ is conflicting for $\boldsymbol{M}$.
c) For any elementary implication $a \Rightarrow b_i$ for $\boldsymbol{M}$ the assignment $a \cup b_i$ is not conflicting for M. ⊗

The BCP-procedure should be consistent with the vector function Y. Now we consider how BCP and the function Y should be coordinated.

Given a full assignment a to inputs of a block $\boldsymbol{B}$, the assignment $(a, Y(a))$ can be viewed as *permissible* complete assignment to pin variables of the block. The Boolean function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ that is defined as a characteristic function of the set of all permissible complete assignments to pin variables of the block $\boldsymbol{B}$ is called the *permission* (or *characteristic*) *function* of the block. If a block $\boldsymbol{B}$ has no outputs, its permission function $\boldsymbol{f}(\boldsymbol{x})$ is defined as an arbitrary Boolean function depending on variables $\boldsymbol{x}$.



(a)

a	b	c	z	y
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(b)

Fig.1 1-Bit-Adder

Example 1. Consider a 1-bit adder (Fig.1(a)). It has three inputs and two outputs where a is the first addend, b is the second addend, c is the carry input, y is the carry output, and z is the sum. Let $\boldsymbol{x} = (a,b,c)$, $\boldsymbol{y} = (y,z)$, then the vector function $\boldsymbol{y} = Y(\boldsymbol{x})$ of the adder is defined by the table represented in Fig.1(b). The permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ for the block (Fig. 1(a)) is the Boolean function taking value 1 only on the combinations of values for a,b,c, z, and y shown in Fig.1(b). ⊗

Recall that a *clause* (or an elementary disjunction) is a disjunction of literals where a *literal* is a Boolean variable or its negation. From now on, we only consider clauses that do not simultaneously contain a literal and its negation. Given a clause $\boldsymbol{d}$, consider the value assignment a to all variables of the clause such that it is unsatisfied. Then we say that the assignment a is *represented* by the clause $\boldsymbol{d}$. For example, clause $a \vee \neg b \vee c$ represents the assignment $a = 0$, $b = 1$, $c = 0$ (we omit brackets for the sake of simplicity). Note that the empty assignment, i.e. one that contains no value assignment to a variable, is represented by the *empty clause*. Considered as a Boolean function the empty clause is equivalent to the constant 0.

Let b be an elementary assignment, say $b = \{x = d\}$. Then $\neg b$ denotes the elementary assignment that assigns an opposite value to the variable x as opposed to b, i.e. $\neg b = \{x = \neg d\}$. Consider an elementary implication $a \Rightarrow b$. We say that a clause $\boldsymbol{d}$ *represents the elementary implication* $a \Rightarrow b$, if the clause represents the assignment $a \cup \neg b$. An explanation of this terminology is the following. Suppose, $\boldsymbol{d} = a \vee \neg b \vee c$. After assigning $a = 0$ and $b = 1$, the only way of satisfying the clause $\boldsymbol{d}$ is to set c to 1. So, the clause $\boldsymbol{d}$ provides a way to derive $c = 1$ under the assignment $a = 0$, $b = 1$. This kind of value derivation is used in the standard BCP-procedure for CNFs, which is specified in Section 3.

A clause $\boldsymbol{d}$ is called *implicate* of a Boolean function $\boldsymbol{f}$, if $\boldsymbol{f}$ implies $\boldsymbol{d}$, i.e. $(\boldsymbol{f} \rightarrow \boldsymbol{d}) = 1$.

***Axiom* 5**. Let a be a partial assignment to the pin variables of a block $\boldsymbol{B}$ and let $b = \mathrm{BCP}(a)$.

a) If a is classified to be conflicting under the BCP-procedure of the block, then the clause representing the assignment a is an implicate of the permission function $\boldsymbol{f}$ of the block.

b) If a implies an assignment b, then for each elementary assignment $b_i \in b$ the clause representing the elementary implication $a \Rightarrow b_i$ is an implicate of the permission function $\boldsymbol{f}$ of the block. ⊗

Discussion: A semantic interpretation of conflicting and implying assignments is the following one. Suppose a Boolean function $j(\boldsymbol{x})$ is tested to be constantly 0. Let $\boldsymbol{N^1}$ and $\boldsymbol{N^0}$ be its On-set and Off-set respectively. Let a be a partial assignment to the variables $\boldsymbol{x}$, and let $Cube(a)$ be the set of all complete assignments containing a. Let $\boldsymbol{P}$ be a procedure which tries to find a counterexample, i.e. a complete assignment $a' \in \boldsymbol{N^1}$. Let $\boldsymbol{P}$ first generate some partial assignments a and then try to extend them to find a counterexample $a' \in Cube(a)$. From the point of view of this procedure, an assignment a is conflicting, if $Cube(a) \subseteq \boldsymbol{N^0}$, because there is no counterexample $a' \in Cube(a)$. However, if $Cube(a) \cap \boldsymbol{N^1} \neq \varnothing$, the assignment a can not be classified as conflicting, because it still can be extended to a counterexample. Thus, an assignment can be classified as conflicting only, if it can be represented by an implicate of $j(\boldsymbol{x})$. Note that permission functions play the role of the function $j(\boldsymbol{x})$ when solving SAT-instances in our theory. (This is formulated in Lemma 25 given in Appendix more precisely.)

On the other hand, an elementary implication $a \Rightarrow b_i$ is correct, if the assignment $a \cup \neg b_i$ is potentially conflicting, i.e. $Cube(a \cup \neg b_i) \subseteq \boldsymbol{N^0}$. We say "potentially conflicting", because we don't postulate that a model must recognize conflicts.

Note, the implication $a \Rightarrow b_i$ can be logically deduced from the fact that the assignment $g = a \cup \neg b_i$ is represented by an implicate of the function $j(\boldsymbol{x})$. Let $a = \{a = 1, b = 1\}$ and $\neg b_i = \{c = 0\}$, and let the assignment $g = a \cup \neg b_i$ be represented by an implicate of $j(\boldsymbol{x})$. Then $g = \{a = 1, b = 1, c = 0\}$ is potentially conflicting, i.e. all complete assignments satisfying the condition $(a = 1) \wedge (b = 1) \wedge (c = 0)$ are falsifying $j(\boldsymbol{x})$. To avoid this potential conflict we should consider assignments satisfying the complemented condition $\neg((a = 1) \wedge (b = 1) \wedge (c = 0))$ which is equivalent to $\neg((a = 1) \wedge (b = 1)) \vee (c = 1)$. By using $(\neg A \vee x) \Leftrightarrow (A \Rightarrow x)$, the latter formula can be formally transformed into $((a = 1) \wedge (b = 1)) \Rightarrow (c = 1)$, i.e. $a \Rightarrow b_i$. ⊗

A block without outputs is called *block-constraint* (as opposed to a *normal* block having at least one output). We need block-constraints in our theory to describe constraints over variables of the system under consideration. A block-constraint having a permission function $\boldsymbol{f}(\boldsymbol{x})$ can classify assignments to its inputs like a normal block and this classification must correlate with the function $\boldsymbol{f}(\boldsymbol{x})$ in accordance to Axiom 5.

***Axiom* 6**. A model $\boldsymbol{M}$ of a normal block $\boldsymbol{B}$ simulates the vector function Y of the block, i.e. for each full value assignment a to the input variables of a block $\boldsymbol{B}$, the BCP-procedure deduces a full value assignment b to the output variables of the block, such that $b = Y(a)$. ⊗

Axioms 5 and 6 postulate the consistency of the BCP-procedure and the vector function Y of a block.

3 CNF-Based Block Models

Recall that that CNF is defined as conjunction of clauses. We say that a CNF $\boldsymbol{C}(\boldsymbol{x})$ represents a Boolean function $\boldsymbol{f}(\boldsymbol{x})$, if $\boldsymbol{C}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x})$. In this section, we show that a CNF representing the permission function $\boldsymbol{f}$ of a block $\boldsymbol{B}$ is a model of the block under the BCP-procedure for CNF's described below.

Given a CNF C and a partial value assignment a to variables of the CNF, BCP(a) is defined by the following procedure. Let C^* denote the current CNF, and let g be the current assignment, and let b be the current implied assignment. At the beginning, let $C^* = C$, $g = a$, $b = \varnothing$. The main cycle of the procedure is started with making all elementary assignments represented in g. If a clause is equal to 0 under g, then the procedure reports a conflict, sets b to $\varnothing$, and stops. Otherwise, all satisfied clauses and literals taking the values 0 under g are removed from the formula, i.e. the latter is transformed to a new current CNF C^*. If there is a *unit* clause in C^*, i.e. a clause containing exactly one literal, say $\neg x$, the assignment satisfying the literal is formed and considered as a new current assignment g (so, we have $g = \{x = 0\}$ for $\neg x$). The assignment g is added to b, i.e. $b := b \cup g$. After that the main cycle is repeated. If there is no unit clause, the procedure stops. In this case, if $b \neq \varnothing$, a is classified to be implying b, otherwise, a is not conflicting or implying. We call the outlined procedure *CNF-BCP*.

Below we show that a CNF representing the permission function $\boldsymbol{f}$ of a block $\boldsymbol{B}$ is a model of the block under *CNF-BCP*, followed by an example. For subsequent considerations, we use the notation (a, b) for $a \cup b$ where a and b are partial assignments and call (a, b) a *pattern*.

Lemma 1. For any full value assignment a to the input variables of a normal block $\boldsymbol{B}$ there is exactly one full assignment b to the output variables, such that $\boldsymbol{f}(a, b) = 1$ where $\boldsymbol{f}$ is the permission function of the block $\boldsymbol{B}$.

Proof : All proofs and other Lemmas are presented in Appendix 1. ⊗

Theorem 1. Any implicate of the permission function of a normal block contains at least one output variable of the block. ⊗

Theorem 2. A CNF representing the permission function $\boldsymbol{f}$ of a block $\boldsymbol{B}$ is a model of the block under *CNF-BCP*. ⊗

The following example helps to clarify Theorem 2.

Example 2. Consider an AND gate implementing the Boolean function $y = a \wedge b$. The permission function $f(a,b,y)$ of the gate takes value 1 on the patterns presented in the table of Fig. 2. The CNF $\boldsymbol{C} = (a \vee \neg y) \wedge (b \vee \neg y) \wedge (\neg a \vee \neg b \vee y)$ represents the function $f(a,b,y)$. According to Theorem 2 the CNF $\boldsymbol{C}$ is a model in our framework. In this case it is the same as the commonly used model for constructing SAT-instances in the domain of formal verification [28].⊗

a	b	y
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 2: On-set of the permission function $f(a,b,y)$ of an AND gate

According to Theorem 2 it is easy to show that conventional CNF representations of gates currently used in EDA domain are models in our framework.

4 Implicativity

In this section, we introduce the notion of implicativity and show that for each block there is always a model with maximal implicativity.

Implicativity of a model $\boldsymbol{M}$ for a block is defined as the number of all conflicting and implying partial assignments to its pin variables. Implicativity can be measured in a straightforward manner by simulation (i.e. by running the BCP-procedure for all possible partial assignments to pin variables).

Now we consider a model that has maximal implicativity among all possible implementations of a Boolean vector function $Y(\boldsymbol{x})$.

We say that a clause $\boldsymbol{d_2}$ *covers* a clause $\boldsymbol{d_1}$, if $\boldsymbol{d_1}$ is a part of $\boldsymbol{d_2}$ (in particular, $\boldsymbol{d_1}$ can be equal to $\boldsymbol{d_2}$). For example $a \vee \neg b \vee c$ covers $a \vee \neg b$. It is easy to see that $\boldsymbol{d_2}$ covers a clause $\boldsymbol{d_1}$ iff $\boldsymbol{d_1} \rightarrow \boldsymbol{d_2} = 1$, i.e. $\boldsymbol{d_1}$ implies $\boldsymbol{d_2}$. A clause $\boldsymbol{d_1}$ is called a prime implicate of a Boolean function $\boldsymbol{f}$, if no other implicate $\boldsymbol{d_2}$ of the function $\boldsymbol{f}$ is covered by $\boldsymbol{d_1}$. It follows from the definition that for any implicate $\boldsymbol{d_2}$ of a function $\boldsymbol{f}$ some part $\boldsymbol{d_1}$ of $\boldsymbol{d_2}$ is a prime implicate of $\boldsymbol{f}$. A CNF that consists of all prime implicates of permission function $\boldsymbol{f}$ is called *characteristic CNF* of $\boldsymbol{f}$ and is denoted $\boldsymbol{C}^{\cdot}$. A characteristic CNF is unique for $\boldsymbol{f}$ and may be considered as normal form for $\boldsymbol{f}$.

Theorem 3. Given a block $\boldsymbol{B}$, the characteristic CNF $\boldsymbol{C}^{\cdot}$ of the permission function $\boldsymbol{f}$ of $\boldsymbol{B}$ is a model with maximal implicativity. ⊗

We say that two clauses $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$ are *orthogonal by a variable x*, if the clauses contain opposite literals of the variable (for example $\boldsymbol{c_1}$ contains x, and $\boldsymbol{c_2}$ contains $\neg x$). *Resolution* can be defined as an operation over two clauses $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$ that are orthogonal by exactly one variable x. Let $\boldsymbol{c_1} = \boldsymbol{d_1} \vee x$ and $\boldsymbol{c_2} = \boldsymbol{d_2} \vee \neg x$ where $\boldsymbol{d_1}$ and $\boldsymbol{d_2}$ are some nonorthogonal clauses (that can be empty). The result of resolving the clauses $\boldsymbol{c_1}$ and $\boldsymbol{c_2}$ is the clause $\boldsymbol{d_1} \vee \boldsymbol{d_2}$ that is called *resolvent* or product of resolution. For example, resolution of the clauses $a \vee b \vee c$ and $\neg a \vee b \vee d$ provides the resolvent $b \vee c \vee d$.

To check whether a CNF C contains all its prime implicates one can test whether there exists a resolvent of any two clauses of the CNF C that is not already presented in C and does not cover any clause of C. If there is no such resolvent, then the CNF C contains all its prime implicates.

Example 3. Consider the model of an AND gate discussed in Example 2. It is easy to check that no pair of clauses can be resolved in the CNF $\boldsymbol{C} = (a \vee \neg y) \wedge (b \vee \neg y) \wedge (\neg a \vee \neg b \vee y)$, because any two clauses are orthogonal by two variables or by none. Thus, the CNF $\boldsymbol{C}$ is itself the characteristic CNF $\boldsymbol{C}^{\cdot}$ of the permission function $f(a,b,y)$. Hence, the considered standard model for AND gate has maximal implicativity. ⊗

It can be shown that commonly used CNF descriptions for the gates implementing elementary Boolean functions are models with maximal implicativity in our framework. However, this encouraging picture immediately disappears as soon as one considers a few gates connected within a combinatorial circuit, as shown in the example below.

Example 4. Consider a circuit of two AND gates having the same input variables (Fig. 3). The conventional CNF of the circuit is the conjunction of the gate CNFs: $\boldsymbol{C} = (a \vee \neg y) \wedge (b \vee \neg y) \wedge (\neg a \vee \neg b \vee y) \wedge (a \vee \neg z) \wedge (b \vee \neg z) \wedge (\neg a \vee \neg b \vee z)$. Given the assignment $y = 0$, the procedure *CNF-BCP* does not fix a conflict or produces any implicate for $\boldsymbol{C}$. At the same time, resolving $\neg a \vee \neg b \vee y$, and $a \vee \neg z$, and $b \vee \neg z$ results in the resolvent $y \vee \neg z$ that can be added to $\boldsymbol{C}$, and that will provide the implying assignment $z = 0$ under $y = 0$. Thus, the conventional CNF $\boldsymbol{C}$ has not the maximal implicativity.

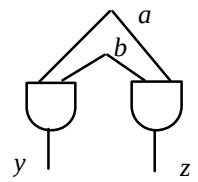


Fig.3 Equivalent gates

5 Observable CNF

Implicativity can be considered as an observable characteristic of a model. Suppose we would like to construct a CNF simulating the model $\boldsymbol{M}$ of a block $\boldsymbol{B}$. Such a CNF is called *observable* and is constructed as follows.

Consider the exhaustive simulation of all possible partial value assignments a to pin variables of the block $\boldsymbol{B}$ (a more efficient procedure is provided in Section 8.1). At the beginning of the procedure, the constructed CNF $\boldsymbol{C}^*$ is empty. If an assignment a is recognized by the model $\boldsymbol{M}$ to be conflicting, we add the clause representing a to $\boldsymbol{C}^*$. If an assignment a is classified by the model $\boldsymbol{M}$ to be implying an assignment b, then for each elementary assignment $b_i \in b$ we add a clause $\boldsymbol{c}$ representing elementary implication $a \Rightarrow b_i$ to $\boldsymbol{C}^*$. After finishing the simulation process, clauses covering other clauses of $\boldsymbol{C}^*$ are removed from $\boldsymbol{C}^*$ one after another. The resulting CNF $\boldsymbol{C}^\blacklozenge$ is the *observable* CNF for the model $\boldsymbol{M}$. According to construction an observable CNF $\boldsymbol{C}^\blacklozenge$ is unique for a model (because the covering relation on the set $\boldsymbol{C}^*$ forms a lattice, and only its minimal elements are kept).

Example 5. Let a block have the CNF $\boldsymbol{C} = (a \vee b \vee c) \wedge (a \vee b \vee \neg c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c)$ as model. In Fig. 4, the results of exhaustive simulation of the model are presented. The second row contains the CNF $\boldsymbol{C}^*$ after finishing simulation. Each cell in the row lists literals (in a column-wise manner) of a clause of $\boldsymbol{C}^*$ which were obtained under the assignment in the same column of the fist row. For example under $b = 0$, $c = 0$, the value $a = 1$ is deduced from the first clause of the CNF $\boldsymbol{C}$. The clause representing the elementary implication $(b = 0, c = 0) \Rightarrow (a = 1)$ is $a \vee b \vee c$. Similarly, the assignment $a = 0$, $c = 0$ is conflicting, which is represented by the clause $a \vee c$. The observable CNF is shown in the third row, i.e. $\boldsymbol{C}^\blacklozenge = (a \vee c) \wedge (a \vee \neg c) \wedge (a \vee b) \wedge (a \vee \neg b)$. ⊗

a b c	- - -	- - 0	- - 1	- 0 -	- 0 0	- 0 1	- 1 -	- 1 0	- 1 1	0 - -	0 - 0	0 - 1	0 0 -	0 0 0	0 0 1	0 1 -	0 1 0	0 1 1	1 - -	1 - 0	1 - 1	1 0 -	1 0 0	1 0 1	1 1 -	1 1 0	1 1 1
$\boldsymbol{C}^*$					a b c	a b $\neg c$		a $\neg b$ c	a $\neg b$ $\neg c$		a c	a $\neg c$	a b 	a b c	a b $\neg c$	a $\neg b$ 	a $\neg b$ c	a $\neg b$ $\neg c$									
$\boldsymbol{C}^\blacklozenge$											a c	a $\neg c$	a b 			a $\neg b$ 											

Fig 4. Constructing an observable CNF

Example 5 also shows that the observable CNF of a model needs not to be equal to its characteristic CNF, because $\boldsymbol{C}^\blacklozenge = (a \vee c) \wedge (a \vee \neg c) \wedge (a \vee b) \wedge (a \vee \neg b)$ but $\boldsymbol{C}^{\cdot} = a$ in the example above.

Now we show that the observable CNF of a model provides an over approximation of its implicativity.

Let $\boldsymbol{M}$ and $\boldsymbol{M}'$ be models of the same block $\boldsymbol{B}$. We say that $\boldsymbol{M}$ and $\boldsymbol{M}'$ are *observably coherent*, if for any partial value assignment a to the block pins they have identical reactions, i.e.:

1. They have no disagreement in classifying a to be conflicting or not, or to be implying.
2. If a is implying, both models deduce the same additional assignment b to the block pins.

Theorem 4. Let CNF $\boldsymbol{C}'$ contain two clauses $\boldsymbol{c}'$ and $\boldsymbol{c}$ where $\boldsymbol{c}'$ covers $\boldsymbol{c}$. Let $\boldsymbol{C}$ be a CNF obtained from $\boldsymbol{C}'$ after removing $\boldsymbol{c}'$. If $\boldsymbol{C}'$ is a model of a block $\boldsymbol{B}$ under *CNF-BCP*, then $\boldsymbol{C}$ is observably coherent to $\boldsymbol{C}'$ under *CNF-BCP*. ⊗

Let $\boldsymbol{M}$ and $\boldsymbol{M}'$ be models of the same block $\boldsymbol{B}$. We say that $\boldsymbol{M}$ *observably covers* $\boldsymbol{M}'$, if for any partial value assignment a to the block pins, observable reaction of both models satisfies the conditions:

1. If $\boldsymbol{M}'$ classifies a to be conflicting, $\boldsymbol{M}$ provides the same classification,
2. If a is implying for $\boldsymbol{M}'$ and $\boldsymbol{M}'$ deduces an additional assignment b to the block pins under a, then a is conflicting for $\boldsymbol{M}$ or a is implying and $\boldsymbol{M}$ generates an additional assignment g under a, such that $b \subseteq g$.

Theorem 5. Let $\boldsymbol{C}^{\blacklozenge}$ be the observable CNF for a model $\boldsymbol{M}$ of a block $\boldsymbol{B}$, then $\boldsymbol{C}^{\blacklozenge}$ (under *CNF-BCP*) observably covers $\boldsymbol{M}$. ⊗

Note that we cannot confirm that the observable CNF for a model is observably coherent to the model. One of the reasons is that the BCP-procedure does not simulate a contraposition law, i.e. if under assignment $x = 0$ we deduce $y = 0$ by using *CNF-BCP*, we cannot deduce value $x = 1$ under the assignment $y = 1$ in the general case. For example, consider CNF $(x \vee a) \wedge (x \vee b) \wedge (\neg a \vee \neg b \vee \neg y)$. Making assignment $x = 0$ and running *CNF-BCP*($x = 0$) we deduce $y = 0$. But *CNF-BCP*($y = 1$) does not provide a value for the variable x.

On the other hand, if we learn from the first experiment that has exposed the observable implication $x = 0 \Rightarrow y = 0$, or to put it more precisely, if we add the clause $x \vee \neg y$ simulating derivation of $y = 0$ under $x = 0$ into the model, we immediately provide an ability to deduce backward implication $y = 1 \Rightarrow x = 1$ by the model under *CNF-BCP*($y = 1$). Therefore, the observable CNF $\boldsymbol{C}^{\blacklozenge}$ for a model $\boldsymbol{M}$ is able to deliver backward implications that are not exposed under *BCP* in the model $\boldsymbol{M}$, and we cannot confirm that the observable CNF $\boldsymbol{C}^{\blacklozenge}$ (under *CNF-BCP*) is observably coherent to the model $\boldsymbol{M}$.

Note that as soon as we add a clause providing derivation of a backward implication in a model we increase implicativity of the latter. In our example, the assignment $y = 1$ will be recognized as implying, whereas before adding the clause $x \vee \neg y$ it was not classified as conflicting or implying.

Our brief discussion suggests that the notion of implicativity is of fundamental importance as well as the idea of increasing implicativity of a model used in SAT-solvers. The next section provides a theoretical foundation for this point of view.

6 Fundamental Theorem on Hierarchical SAT-Solving

In this section we show that for a normal block with maximal implicativity the considered SAT-solving problem is trivial (Theorem 6). In the sequel, we consider a CNF also as a set of clauses, where $\boldsymbol{c} \in \boldsymbol{C}$ denotes that a clause $\boldsymbol{c}$ is contained in a CNF $\boldsymbol{C}$, and $\boldsymbol{C} \subseteq \boldsymbol{C}'$ denotes that all the clauses of a CNF $\boldsymbol{C}$ belong to a CNF $\boldsymbol{C}'$, and so on.

Theorem 6. Let $\boldsymbol{M}$ be a model of a normal block $\boldsymbol{B}$, and let $\boldsymbol{M}$ have maximal implicativity. Let $\boldsymbol{y}$ be an output of the block $\boldsymbol{B}$. The output $\boldsymbol{y}$ implements a constant Boolean function d where $d \in \{0,1\}$ iff the elementary assignment $\neg a = \{ \boldsymbol{y} = \neg d \}$ is conflicting or the empty assignment $g = \varnothing$ implies $a = \{ \boldsymbol{y} = d \}$ for the model $\boldsymbol{M}$. ⊗

Note to prove Theorem 6 we use neither Axiom 4 nor Axiom 6. Thus, Theorem 6 is correct in a more general theory which doesn't contain these axioms.

It is a well known fact that a CNF containing all prime implicates of the Boolean function realized by the CNF is easily testable for satisfiability under *CNF-BCP*. Theorem 6, particularly, shows that in this case it is also trivial to check whether a CNF representing a permission function is equivalent to a unit clause. Theorem 6 is of fundamental importance because it holds for any model that satisfies our axiomatic system and it relates hardness of SAT-solving with the notion of implicativity.

7 Consistent Models

One might want to show that any two models with maximal implicativity for the same block are observably coherent. However, this is not correct in the general case.

Example 6. Consider again the model of an AND gate (Fig. 2). It is proven (in Example 3) that CNF $(a \vee \neg y) \wedge (b \vee \neg y) \wedge (\neg a \vee \neg b \vee y)$ is the characteristic CNF $\boldsymbol{C}^{\cdot}$ of the permission function of the gate. Thus, it is a model with maximal implicativity. Suppose that we perform exhaustive simulation over all partial assignments to the pin variables of the gate. We record the assignments in a table and mark each assignment with the reaction of the gate: 1) if the assignment is inconsistent with the gate functionality, we mark the assignment as conflicting; 2) if the assignment is not conflicting and new values to the pin variables are deduced, we record the produced assignment. A part of the table is presented in Fig. 5(a). We can define a table based model for AND gate as follows: given an assignment, look into the table and deliver the recorded reaction. This "table model" will be observably coherent to the CNF $\boldsymbol{C}^{\cdot}$ model by construction.

a	*b*	*y*	marks
-	-	1	$a = 1, b = 1$
0	-	1	conflict
-	0	1	conflict
-	-	0	no reaction
-	1	1	$a = 1$
.	.	.	. . .

(a)

a	*b*	*y*	marks
-	-	1	$a = 1$
0	-	1	conflict
-	0	1	conflict
-	-	0	no reaction
-	1	1	$a = 1$
.	.	.	. . .

(b)

Fig. 5. Table models for AND gate

Suppose, we change the first row of the table as it is shown in Fig. 5 (b). The new table again is a model of the gate, because: a) it reproduces the input to output functionality of the gate, b) it classifies all the partial assignments exactly as the first table based model does, and c) it preserves the monotony of the classification. Indeed, the only modification is in the first row. Both models recognize the partial assignment $y = 1$ to be implying. Classification of $y = 1$ is still monotone in the second model, as $y = 1 \Rightarrow a = 1$ while $\{b = 0, y = 1\}$ is conflicting and $\{b = 1, y = 1\}$ is implying. Thus, both models have maximal implicativity. However, they are not observably coherent, since the first model produces $a = 1$, $b = 1$ under the assignment $y = 1$, but the second one deduces $a = 1$ only.

In a sense, the second model is not logically tight. It recognizes $b = 0$, $y = 1$ to be conflicting assignment and $y = 1$ to be implying assignment, but it does not produce the implication $y = 1 \Rightarrow b = 1$ that "logically closes" the considered situation. ⊗

Now we introduce a notion of a consistent model and show that consistent models for the same normal block are observably coherent and have maximal implicativity.

We call a model $\boldsymbol{M}$ of a block $\boldsymbol{B}$ *consistent*, if the following conditions hold:

1) For any elementary implication $a \Rightarrow b_i$ for $\boldsymbol{M}$, the assignment $a \cup \neg b_i$ is conflicting for $\boldsymbol{M}$.
2) If an assignment a is conflicting for $\boldsymbol{M}$, then for any elementary assignment $b_i \in a$ the assignment $a' = a \setminus b_i$ is conflicting or $a' \Rightarrow \neg b_i$ for $\boldsymbol{M}$.

The model from the Table 5.*b* is not consistent, as the assignment $\{g = 1, b = 0\}$ is conflicting, however $y = 1$ does not imply $b = 1$. As we will see in the sequel many practical models are consistent.

Theorem 7. The characteristic CNF $\boldsymbol{C}^{\cdot}$ of a block $\boldsymbol{B}$ is a consistent model $\boldsymbol{M}$ under *CNF-BCP*. ⊗

A model $\boldsymbol{M}$ for a block $\boldsymbol{B}$ is called *recognizing maximal conflicts*, if any complete assignment g to the pin variables of $\boldsymbol{B}$ which falsifies the permission function $\boldsymbol{f}$ of $\boldsymbol{B}$ is conflicting for $\boldsymbol{M}$.

Theorem 8. A consistent model $\boldsymbol{M}$ of a normal block $\boldsymbol{B}$ is recognizing maximal conflicts. ⊗

Theorem 9. Any consistent and recognizing maximal conflicts model $\boldsymbol{M}$ of a block $\boldsymbol{B}$ has maximal implicativity. ⊗

Theorem 10. Any consistent model $\boldsymbol{M}$ of a normal block $\boldsymbol{B}$ has maximal implicativity. ⊗

Theorem 11. Any two consistent models with maximal implicativity for a block $\boldsymbol{B}$ are observably coherent. ⊗

Theorem 12. Any two consistent models for a normal block $\boldsymbol{B}$ are observably coherent. ⊗

It is possible (as in Example 6) that two models of a block provide identical classification of partial assignments however the same implying assignment can provoke different numbers of elementary implications in the models. To compare models having close implicativity one can use the following notion. *Strong implicativity* is the total number of conflicting assignments and elementary implications of the model under consideration. Strong implicativity can be measured by simulation similarly to implicativity as discussed in the next section.

Now we consider relations between notions of implicativity, strong implicativity, model's consistency and coherency.

Theorem 13. Any model $\boldsymbol{M}$ with maximal strong implicativity for a block $\boldsymbol{B}$ has maximal implicativity. ⊗

Theorem 14. Any model $\boldsymbol{M}$ with maximal strong implicativity for a block $\boldsymbol{B}$ is consistent. ⊗

Theorem 15. Any two models with maximal strong implicativity for a block $\boldsymbol{B}$ are observably coherent. ⊗

Theorem 16. Any consistent model $\boldsymbol{M}$ with maximal implicativity for a block $\boldsymbol{B}$ has maximal strong implicativity. ⊗

Theorem 17. A model $\boldsymbol{M}$ of a normal block $\boldsymbol{B}$ is consistent iff it has maximal strong implicativity. ⊗

Thus, by constructing a consistent model for a normal block maximal strong implicativity and maximal implicativity of the model are reached.

In general, strong implicativity is a more refined measure for model comparison than implicativity. However, implicativity is a simpler notion, as it does not take into account the structure of implications deduced by the model. In the sequel, we focus our discussion around this notion of implicativity.

8 Estimating the Implicativity of Models

Implicativity of a block can be measured by exhaustively simulating all possible partial value assignments to pin variables of the block and running the block's BCP-procedure for each assignment. Each time the BCP-procedure recognizes an affecting assignment to be conflicting or implying, a counter of implicativity is incremented by 1 (in the beginning of the simulation the counter is equal to 0).

Regarding the number of possible partial assignments and performance of modern computers, this trivial method is restricted to blocks having about 16 pin variables. It is useful to have a way of estimating implicativity for blocks having more pins. On the other hand, it is very important for any block to have a way of comparing the current implicativity with the maximal one for the block.

In this section, we propose two procedures which help us to estimate both the current and the maximal implicativity of a block $\boldsymbol{B}$. The first procedure constructs the observable CNF $\boldsymbol{C}^{\cdot\cdot}$ for $\boldsymbol{B}$. According to Theorem 5 the observable CNF $\boldsymbol{C}^{\cdot\cdot}$ provides an over approximation of its implicativity. The second procedure constructs the characteristic CNF $\boldsymbol{C}^{\cdot}$ that has maximal implicativity among all possible block models (according to Theorem 3). By comparing $\boldsymbol{C}^{\cdot\cdot}$ and $\boldsymbol{C}^{\cdot}$ the implicativity of the current block's model can be estimated. At the same time, investigating the difference between $\boldsymbol{C}^{\cdot}$ and $\boldsymbol{C}^{\cdot\cdot}$ one can find a way to increase the implicativity of the model. (In principle, the current model $\boldsymbol{M}$ of the block $\boldsymbol{B}$ can be replaced with the characteristic CNF $\boldsymbol{C}^{\cdot}$ guaranteeing maximal implicativity. However, if $\boldsymbol{C}^{\cdot}$ is large, a more concise model than $\boldsymbol{C}^{\cdot}$ has to be used).

We consider only basic algorithms for constructing $\boldsymbol{C}^{\cdot\cdot}$ and $\boldsymbol{C}^{\cdot}$ and leave implementations, optimizations, and compacting data structures for further research.

8.1 Constructing the Observable CNF by Experimenting with a Model

In this section, we describe the procedure *CONSTRUCT_OBS* for constructing the observable CNF $\boldsymbol{C}^{\cdot\cdot}$ of the current model $\boldsymbol{M}$ of a block $\boldsymbol{B}$. The procedure considers the set of all partial assignments to the pin variables of the block $\boldsymbol{B}$ by constructing a ternary tree and traversing it. In contrast to the procedure of exhaustive simulation it does not consider conflict assignments containing (covering) conflict assignments of smaller size.

Let $\boldsymbol{U}$ be a set of pin variables of the block. A variable $u \in \boldsymbol{U}$ can take a value from the set {0,1,-, x}. At the beginning of the procedure, all variables from $\boldsymbol{U}$ have value "x". The value "x" of u means that u is "free to branch on" and can take any value 0, 1, or "-". The value "-" of u is interpreted during

the procedure such that the variable cannot be used to branch on and it is considered as unassignable to a definite value (0 or 1) under decision making. If variable u has value "-" or "x" and a necessary assignment 0 or 1 is deduced for u by the BCP-procedure of the block $\boldsymbol{B}$, then this assignment will be used until BCP is finished. After that it will be ignored and the indefinite value of u will be recovered. At each node of the search tree the procedure performs three steps:

1) *Decision making*: It selects a free variable u to branch on (any free variable can be selected) or backtracks, if there is no free variable;
2) *Branch selection*: It selects a value from the set {0,1,-} to be assigned to the variable u. Firstly, values $u = 0$ and $u = 1$ are selected in any order, and then "-". Values $u = 0$ and $u = 1$ are called *definite decision assignments*. After the branch "-" has been examined, the procedure makes assignment u = "x" and backtracks;
3) *BCP*: It runs the *BCP-procedure* of the model $\boldsymbol{M}$ under the current assignment a to variables of the set $\boldsymbol{U}$ (variables having value "-" or "x" are considered as unassigned). There are three cases:
   a) *BCP* classifies the assignment a to be conflicting for $\boldsymbol{M}$. Then the clause representing a is added to the constructed observable CNF $\boldsymbol{C}^{\cdot\cdot}$ (which is empty in the beginning of). The current branch of the search tree is considered to be finished, and the procedure performs step 2 of selecting the next branch.
   b) *BCP* classifies the assignment a to be implying an assignment b for $\boldsymbol{M}$. Then for each elementary implication $a \Rightarrow b_i$ where $b_i \in b$ the clause representing implication $a \Rightarrow b_i$ is added to the constructed observable CNF $\boldsymbol{C}^{\cdot\cdot}$. After that the procedure makes the next decision (step 1). (Recall that, deduced assignment b is ignored).
   c) *BCP* does not recognize the assignment a as conflicting or implying. Then the procedure continues with step 1.

After exhausting the decision making, every clause that covers some other clause of the $\boldsymbol{C}^{\cdot\cdot}$ is removed from $\boldsymbol{C}^{\cdot\cdot}$.

***Remark*:** After identifying an assignment a to be implying in step (b) the search is continued in depth because it is necessary in the general case. Suppose a block with has two inputs a, b and two outputs x, y. Let $a = 1$ imply $x = 1$ and $a = 1$, $b = 0$ imply $x = 1$, $y = 0$. Suppose the current assignment is $a = 1$ (and all other variables are free), then the search has to continue in depth to recognize the implication $a = 1, b = 0 \Rightarrow x = 1, y = 0$. ⊗

Theorem 18. The procedure *CONSTRUCT_OBS* constructs the observable CNF $\boldsymbol{C}^{\cdot\cdot}$ for the block $\boldsymbol{B}$. ⊗

8.2 Constructing the Characteristic CNF by Experimenting with a Model

Given the characteristic function $\boldsymbol{f}$ of a block $\boldsymbol{B}$, all prime implicates of $\boldsymbol{f}$ must be generated to construct the characteristic CNF $\boldsymbol{C}^{\cdot}$. Using the procedure *CONSTRUCT_OBS*, a representation of $\boldsymbol{f}$ form of the observable CNF $\boldsymbol{C}^{\cdot\cdot}$ can be found. After that, the characteristic CNF $\boldsymbol{C}^{\cdot}$ can be computed theoretically by using classical method of Blake-Poretski [29,30]. The method performs resolutions (see Section 4) over clauses of the current CNF and adds resolvents to this CNF until a resolvent can be produced that does not cover a clause already presented in the formula. All clauses that cover some other clauses of the formula are to be removed from the latter. The method is unpractical for large CNFs, however it is important for our consideration because it delivers the following sufficient condition for a CNF to be the characteristic one.

Theorem 19. A CNF representing a permission function $\boldsymbol{f}$ is the characteristic CNF $\boldsymbol{C}^{\cdot}$, if for any two clauses of the CNF that can be resolved their resolvent covers a clause of the CNF. ⊗

Given the observable CNF $\boldsymbol{C}^{\cdot\cdot}$, the characteristic CNF $\boldsymbol{C}^{\cdot}$ can be constructed by using an advanced method of generating all prime implicates, for example [31,32]. In this section, we consider a procedure *CONSTRUCT_CH* that does not use CNF $\boldsymbol{C}^{\cdot\cdot}$ and constructs $\boldsymbol{C}^{\cdot}$ directly by experimenting with the given block $\boldsymbol{B}$. (When the procedure starts to work, the permission function $\boldsymbol{f}$ of the block is unknown.)

The procedure *CONSTRUCT_CH* basically differs from the procedure *CONSTRUCT_OBS* in its learning ability. On the one hand, it has to find prime implicates of the permission function $\boldsymbol{f}$ that are covered by clauses of the observable CNF $\boldsymbol{C}^{\cdot\cdot}$. On the other hand, it has to find prime implicates that are not covered by clauses of $\boldsymbol{C}^{\cdot\cdot}$, if they exist. At last, it has to guarantee that all possible prime implicates are found. It has the following seven constructive differences from the procedure *CONSTRUCT_OBS*.

1. The value "-" of a decision variable or the value "x" of a free variable is to be replaced with a deduced value 0 or 1 as soon as the deduction occurs. However after finishing examination of the current branch, indefinite values "-" and "x" are to be restored according to the restore rule described below.
2. We will distinguish the current *path decision* assignment $a^{\cdot}$ that contains only all definite decision assignments of the path leading from the root of the search tree to the current node $\boldsymbol{N}$, and the current *extended* assignment a that contains all decision and deduced assignments of values 0 and 1 to the variables of the set $\boldsymbol{U}$ at the current state of the procedure.
3. The BCP-procedure is run under the current ***extended*** assignment a.
4. Now the BCP-procedure is *mixed*. Let $\boldsymbol{C}^*$ be the current CNF constructed by the procedure *CONSTRUCT_CH*. In the beginning of the procedure, $\boldsymbol{C}^*$ is empty, and $\boldsymbol{C}^* = \boldsymbol{C}^{\cdot}$ in the end. In step 3, the procedure *CONSTRUCT_CH* runs *BCP* under current extended assignment ***repeatedly over*** the block $\boldsymbol{B}$ and the current CNF $\boldsymbol{C}^*$ until a conflict is encountered or *BCP* (over the block and the CNF) produces no new implication. (It is the *CNF-BCP* procedure described in Section 3 that is run over the current CNF $\boldsymbol{C}^*$).
5. If the mixed *BCP-procedure* recognizes the current extended assignment a to be conflicting, then the clause representing the current ***path decision*** assignment $a^{\cdot}$ is to be added to the current CNF $\boldsymbol{C}^*$.
6. If the mixed *BCP-procedure* recognizes the current extended assignment a to be implying an assignment b, then for each elementary implication $a^{\cdot} \Rightarrow b_i$, where $b_i \in b$ and $a^{\cdot}$ is the current ***path decision*** assignment, the clause representing implication $a^{\cdot} \Rightarrow b_i$ is to be added to the current CNF $\boldsymbol{C}^*$. The deduced assignment b is temporally valid as described below until the current branch traversal is finished.

   Restore rule: Let the current branch be $u = s$ ($s \in \{0,1\}$), and let the current extended assignment be $a = g \cup \{ u = s \}$. Let the mixed *BCP-procedure* deduce a necessary assignment 0 or 1 to a variable v that has the value "-" (or "x") before the branch. As soon as the branch traversal is finished all such deduced necessary assignments are to be erased and indefinite values "-" (respectively "x") are to be restored.
7. *Conflict "inheritance"*. Let $\boldsymbol{N}$ be the current node of the search tree, and let a be the current extended assignment. Let the procedure *CONSTRUCT_CH* branch on a variable u in the node $\boldsymbol{N}$. Let $u = s$ where $s \in \{0,1\}$, and let the assignment $a \cup \{ u = s \}$ be classified to be conflicting by the procedure *CONSTRUCT_CH*. Then the branch $u = s$ of the node $\boldsymbol{N}$ is marked as conflicting. The node $\boldsymbol{N}$ is considered to be conflicting, if both its branches $u = 0$ and $u = 1$ or the branch $u =$ "-" are marked as conflicting. Let the procedure backtrack to the parent node $\boldsymbol{N}'$. If $\boldsymbol{N}$ is a conflicting node, then the branch leading from $\boldsymbol{N}'$ to $\boldsymbol{N}$ is marked as conflicting. In this case, if both branches $u = 0$ and $u = 1$ of N are marked as conflicting, the clause representing the ***path decision*** assignment $a^{\cdot}$ (of the path from the root to the node $\boldsymbol{N}$) is added to the current CNF $\boldsymbol{C}^*$. Thus, the procedure "inherits" conflicts of branches and reduces represented conflict clauses.

Theorem 20. The procedure *CONSTRUCT_CH* delivers the characteristic CNF $\boldsymbol{C}^{\cdot}$ for the block $\boldsymbol{B}$. ⊗

The procedures *CONSTRUCT_OBS* and *CONSTRUCT_CH* can be considered as skeletons for practical algorithms to be developed.

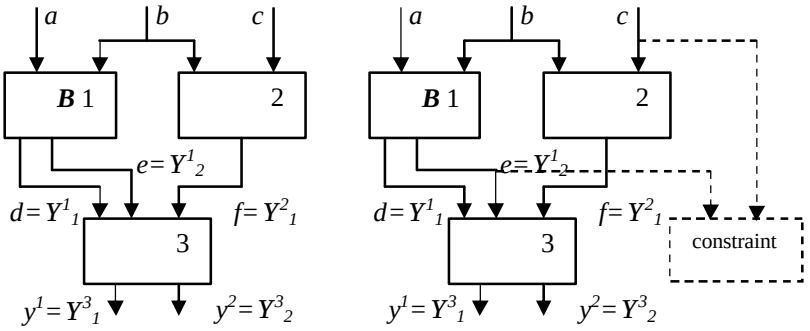
9 A System as a Normal Block

In this section, our goal is to define a model for a system of blocks. We will introduce a *SYSTEM-BCP procedure* and prove that the latter satisfies our axioms. As a result, a system can be considered as a high level block composed of its blocks.

9.1 Axioms 1 and 2

Recall that we only consider systems that can be treated as combinatorial circuits where gates correspond to blocks. First, consider a system containing normal blocks only. Such a *normal system* $\boldsymbol{S}$ corresponds to a directed acyclic graph $\boldsymbol{G}$. Source and sink nodes of $\boldsymbol{G}$ correspond to primary inputs and primary outputs of the system $\boldsymbol{S}$, respectively. The other nodes are set into one to one correspondence to blocks of the system. Each edge of the graph is marked with a Boolean variable. For an outgoing (incoming) edge marked with z of some node corresponding to a block $\boldsymbol{B}$, the variable z must be the output (input) variable of $\boldsymbol{B}$. The variables marking outgoing (incoming) edges of primary

inputs (primary outputs) are called *input* (*output*) *variables* of the system. The other variables are *internal variables* of the system. Input and output variables of a system are its *pin variables*. For the system depicted in Fig. 6, *a*, *b*, *c* are input variables, *d*, *e*, *f* are internal variables, and y^1, y^2 are output variables.



a) normal b) constrained

Fig. 6 A system of 3 normal blocks

Considering the pin variables of a system only results in a "big" block. Thus, Axiom 1 holds for a normal system. (To be more precise, we restrict our consideration to finite systems, i.e. systems containing a finite number of blocks. Since each block has a finite number of pin variables, the system has the same property.)

A normal system corresponds to the Boolean vector function $\boldsymbol{y} = Y(\boldsymbol{x})$ implemented by the system considered as a combinatorial circuit where $\boldsymbol{x}$ is the vector of input variables of the system and $\boldsymbol{y}$ is the vector of output variables of the system.

Formally, the vector function implemented by a normal system can be constructed as a superposition of vector functions implemented by its blocks. The superposition is constructed in reverse topological order, starting from system's outputs. For example, for the system in Fig. 6.a, we have $y^1(a,b,c) = Y^3_1(Y^1_1(a,b), Y^1_2(a,b), Y^2_1(b,c))$, $y^2(a,b,c) = Y^3_2(Y^1_1(a,b), Y^1_2(a,b), Y^2_1(b,c))$.

A *constrained* system (or simply, system) can be obtained from a normal system by adding some block-constraints as depicted in Fig. 6.b. The normal system is called the *normal* (or *basic*) *part* of the constrained system. The vector function implemented by a system is defined to be the same as for its normal part. The permission function of a block-constraint has to satisfy a condition considered in the next section (called fitting axiom) to be compatible with the system's functionality.

9.2 Permission Function

Given a system $\boldsymbol{S}$ implementing a Boolean vector function $\boldsymbol{y} = Y(\boldsymbol{x})$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of input and output variables of the system $\boldsymbol{S}$, respectively, the permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ of the system is defined just as for a block, i.e. $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ is the characteristic function of the set of permissible complete assignments $g = (a, Y(a))$ to pin variables of the system.

Let the normal part of the system $\boldsymbol{S}$ consist of normal blocks $\boldsymbol{B}^1$, $\boldsymbol{B}^2$,…, $\boldsymbol{B}^k$ having permission functions $\boldsymbol{f}^1$, $\boldsymbol{f}^2$,…, $\boldsymbol{f}^k$, respectively, and let $\boldsymbol{z}$ be a vector of internal variables of $\boldsymbol{S}$. *The extended permission function* $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ of $\boldsymbol{S}$ is defined as conjunction of the functions $\boldsymbol{f}^1$, $\boldsymbol{f}^2$,…, $\boldsymbol{f}^k$, i.e. $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \wedge \boldsymbol{f}^i$ $(i = 1,\ldots,k)$.

Now consider an axiom coordinating constraints with a system's functionality.

Fitting axiom. Let $\boldsymbol{B}^c$ be a block-constraint for the system $\boldsymbol{S}$, and let $\boldsymbol{f}^c$ be its permission function. Then $\boldsymbol{f}^c$ must be implied by the extended permission function $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ of the system. ⊗

Due to the fitting axiom the extended permission function of the system is also equal to the conjunction of the permission functions of all system blocks (both normal and constraints).

In this section, we show that the permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ of a system can be obtained from its extended permission function $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ by existential quantification of the latter on internal system variables $\boldsymbol{z}$.

The operator $\$a$ for *existential quantification* of a Boolean function f w.r.t. a variable a is defined as follows: $\$a\ f(x_0,\ldots,a,\ldots,x_n) = f(x_0,\ldots,0,\ldots,x_n) \vee f(x_0,\ldots,1,\ldots,x_n)$. The operator is commutative and associative. Thus, given a set of variables, the existential quantification w.r.t. a set of variables can be performed in any order. We will use a *vector notation* $\$\boldsymbol{z}\ \boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ denoting quantification on all variables of the vector $\boldsymbol{z}$.

Given a Boolean function $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ and a complete assignment g to the variables of the vector $\boldsymbol{z}$, the function $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},g)$ is called a *cofactor* of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ w.r.t. assignment g and is denoted as $\boldsymbol{f}^*_{\boldsymbol{z}=g}$. Let $2^{\boldsymbol{z}}$

be the set of all complete assignments to the variables of the vector $\boldsymbol{z}$. According to the definitions above we have:

$$\exists z\, \boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \vee \boldsymbol{f}^*_{\boldsymbol{z}=\gamma} \quad (\gamma \in 2^{\boldsymbol{z}}). \tag{1}$$

Theorem 21. Let $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ be the permission function and let $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ be the extended permission function of a system $\boldsymbol{S}$. Then $\exists z\, \boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$. ⊗

Thus, a way of constructing the permission function of a system $\boldsymbol{S}$ is to find its extended permission function $\boldsymbol{f}^*$ by formula (1) and then existentially quantify the latter by the internal variables of $\boldsymbol{S}$.

9.3 SYSTEM-BCP Procedure

In this section, we introduce two BCP-procedures for a system. Henceforth we will prove that the procedures satisfy Axioms 3 – 6 of a block model. Thus, a system of blocks can be considered as a model under these procedures.

Let $\boldsymbol{S}$ be a system consisting of blocks $\boldsymbol{B}^j$ each having a model $\boldsymbol{M}^j$ (j = 1,…,k). We define the first procedure *SYSTEM-BCP*(α) for the system $\boldsymbol{S}$ under an assignment α to some variables of the system. Let $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ be vectors of input, output and internal variables of the system $\boldsymbol{S}$, respectively, and let $\boldsymbol{v} = (\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ be the vector of all variables of the system. To define the procedure we take into account the structure of $\boldsymbol{S}$. For each variable $v_i \in \boldsymbol{v}$ let $IN(v_i)$ be a set of all blocks $\boldsymbol{B}^j$ having v_i as input variable.

Let *DEDUCED* be a list of elementary value assignments to variables from $\boldsymbol{v}$. The list is used by *SYSTEM-BCP*(α) and has two pointers *BOTTOM* and *TOP*. At the beginning of the procedure, the list *DEDUCED* is filled with all elementary assignments from α, and after that the pointer *BOTTOM* indicates the first element of the list *DEDUCED*. The pointer *TOP* indicates the last element of the list. The semantics of the list at a current state of the procedure is as follows. All elementary assignments *before* the *BOTTOM* assignment *have already been substituted* into the model. The elementary assignments starting with *BOTTOM* and ending with *TOP* are considered as a queue of *candidates* to be substituted later on.

Let *REASONS* be a list whose elements correspond to the elements of the list *DEDUCED*. If an elementary assignment b_i belongs to the list *DEDUCED*, then the corresponding element of the list *REASONS* contains a *direct reason* of b_i. A direct reason is an assignment defined below. The direct reason of any elementary initial assignment from α is the empty assignment ∅.

On its main cycle the procedure substitutes the "*BOTTOM* assignment", i.e. the assignment indicated by the *BOTTOM* pointer, in one of the following two ways. (After the substitution the *BOTTOM* pointer is increased to indicate the next element of the list *DEDUCED*.)

1) The *BOTTOM* assignment is the empty one, the empty assignment will be applied to each block of the system. (We consider this specific case, because the empty assignment is allowed to affect a block (or a system, if we would like to consider a system as block) in our system of axioms).
2) If the *BOTTOM* assignment is an elementary one, say $v_i = s$, the procedure checks whether there exists the complementary assignment $v_i = \neg s$ before the *BOTTOM* element in the list *DEDUCED*. If there is such an assignment, the procedure classifies α to be conflicting and stops. The pair of assignments ($v_i = s$, $v_i = \neg s$) is considered to be the *direct reason of the conflict*. Otherwise, the substituton operation for the assignment $v_i = s$ is performed separately for each block of the set $IN(v_i)$. Suppose $\boldsymbol{B}^j \in IN(v_i)$, and γ^j is a set of all elementary assignments to the pin variables of the block $\boldsymbol{B}^j$ that are stored *before* the *BOTTOM* assignment in the list *DEDUCED*. We run the procedure *BCP*(γ^j, $v_i = s$) for the block $\boldsymbol{B}^j$. Now there are three cases:
   A) The assignment (γ^j, $v_i = s$) is classified to be conflicting for the block $\boldsymbol{B}^j$. Then the procedure *SYSTEM-BCP*(α) stops and reports that the initial assignment α applied to the pin variables of the system $\boldsymbol{S}$ is conflicting. The assignment (γ^j, $v_i = s$) is called the *direct reason of the conflict*.
   B) The assignment (γ^j, $v_i = s$) is classified to be implying an additional assignment β^j to the pin variables of the block $\boldsymbol{B}^j$. Then all elementary assignments from β^j are pushed into the list *DEDUCED* and after that the pointer *TOP* indicates the last elementary assignment from β^j which was pushed into the list. For each elementary assignments β^j_i from β^j the assignment (γ^j, $v_i = s$) is considered to be *the direct reason of* β^j_i and is stored in the list *REASONS* at the position corresponding to β^j_i.
   C) The assignment (γ^j, $v_i = s$) is not recognized as conflicting or implying for the block $\boldsymbol{B}^j$.

The procedure stops when *BOTTOM* pointer exceeds *TOP*, i.e. all assignments stored in the list *DEDUCED* have been already substituted. If there are additional elementary assignments to the pin

variables of the system $\boldsymbol{S}$ in the list *DEDUCED* which are not contained in the initial assignment a, then the assignment a is classified to be implying the assignment b consisting of these additional elementary assignments.

Note that the considered procedure *SYSTEM-BCP*(a) is conservative in the sense that it "sees" substituted assignments only, i.e. assignments stored in the list *DEDUCED* before the *BOTTOM* element. As a consequence it recognizes conflicts and makes implications with a delay as is in the next example.

Example 7. Consider a system $\boldsymbol{S}$ that has the structure depicted in Fig. 6a). Suppose the initial assignment a is $b = 0, y^1 = 0$. The chronology of running the procedure *SYSTEM-BCP*(a) is presented

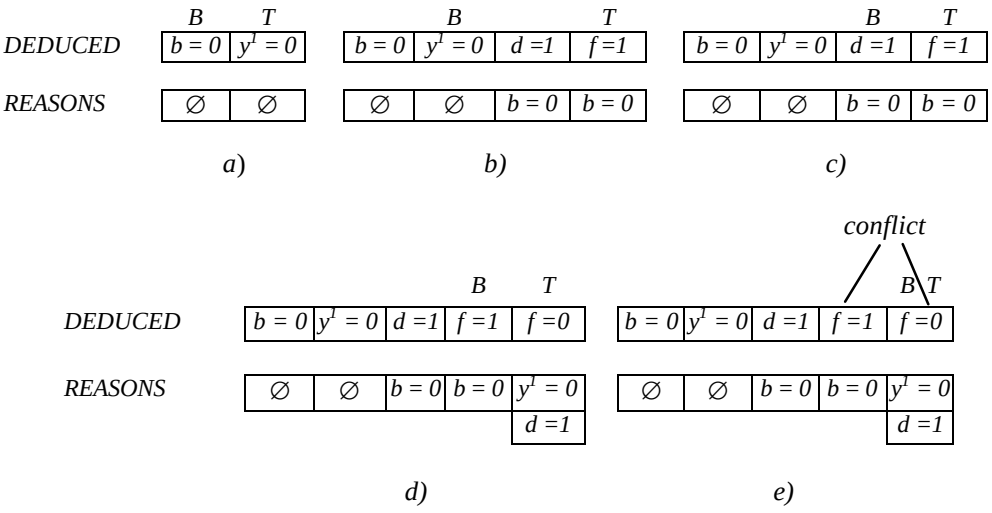


Fig. 7. *SYSTEM-BCP*($b = 0, y^1 = 0$)
a) before substituting $b = 0$; *b*) before substituting $y^1 = 0$; *c*) before substituting $d = 1$;
d) before substituting $f = 1$; *e*) before substituting $f = 0$;

in Fig. 7. Here, the *BOTTOM* and the *TOP* pointers are marked with letters *B* and *T*. The procedure is started with representing the initial assignment $b = 0, y^1 = 0$ in the list *DEDUCED* (Fig. 7*a*). Suppose that substituting $b = 0$ resulted in deduction of the assignment $d = 1$ from block 1 and the assignment $f = 1$ from block 2 (Fig. 6 and Fig. 7*b*). Given $b = 0$, let substituting $y^1 = 0$ produce no new implications (Fig. 7*c*). Given $b = 0$ and $y^1 = 0$, suppose that substituting $d = 1$ produces the assignment $f = 0$ which is implied by block 3 under the partial assignment $y^1 = 0$, $d = 1$ (Fig. 7*d*). Now suppose that substituting $f = 1$ produces no new implication. At the next step the procedure recognizes a conflict, because before the *BOTOM* element $f = 0$ there is the complementary assignment $f = 1$ (Fig. 7e). The pair $f = 0, f = 1$ is treated as the direct reason of the conflict. ⊗

Now we consider a modification of this procedure called *FORCED-SYSTEM-BCP*(a) that eliminates the drawback mentioned above. The first change is as follows. When the procedure adds a new elementary assignment b^j_i to the list *DEDUCED*, it immediately checks whether there is the opposite assignment $\neg b^j_i$ in the list. If the opposite assignment is contained in the list *DEDUCED*, the procedure stops and reports that the initial assignment a is conflicting. In this case, the pair of assignments $b^j_i, \neg b^j_i$ is treated as the direct reason of the conflict.

The idea of checking a pair of opposite assignments in the list *DEDUCED* is known as *early conflict detection* in the domain of CNF SAT-solving and was firstly implemented in the SAT-solver *Limmat* [33].

The second change can be called making *early implications.* Suppose we substitute the assignment $v_i = s$ for a block $\boldsymbol{B}^j \in IN(v_i)$. Instead of assignments stored *before* the *BOTTOM* assignment in the list *DEDUCED* we take into account *all* assignments contained in the list. So, let g^j be a set of elementary assignments to the pin variables of the block $\boldsymbol{B}^j$ except of $v_i = s$ that are stored in the list *DEDUCED*, we run the procedure *BCP*($g^j, v_i = s$) for the block $\boldsymbol{B}^j$.

Example 8. Consider the same system $\boldsymbol{S}$ as in Example 7. The chronology of running the procedure *FORCED-SYSTEM-BCP*(a) is presented in Fig. 8. We see that now the procedure detects a conflict when substituting the assignment $y^1 = 0$. ⊗

Note that a direct reason for a conflict under the both procedures can be of two types. First, it can be a pair of opposite elementary assignments represented in the list *DEDUCED*. Second, it can be a conflicting partial assignment for a block of the system under consideration.

One can find drawbacks in both procedures. For example the first detects conflicts with a delay, while the second one can make attempts to substitute a value which was substituted earlier when the procedure passed through the list *DEDUCED* before the current element (i.e. the procedure can run BCP for a block twice for the same value assignment to the block's pin variables). The two procedures implement opposite approaches between which others can be formulated. For example one can consider a procedure that uses early conflict detection, but not early implications. We will show that

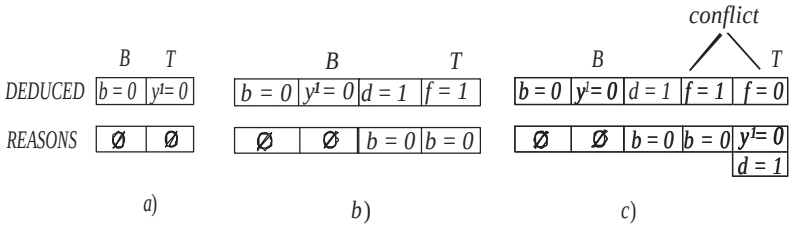


Fig. 8. *FORCED-SYSTEM-BCP*($b = 0, y^1 = 0$)

a) before substituting $b = 0$; *b*) before substituting $y^1 = 0$; *c*) after substituting $y^1 = 0$;

both procedures satisfy our system of axioms. As a consequence the same will be valid for any intermediate version.

9.4 REVERSE-BCP Procedure

For discussion in Sections 9.4 and 9.5, we need to extend the notion of assignment allowing it to contain opposite value assignments to the same variable. For example $g = \{f = 0, f = 1\}$. The clause representing g is $f \vee \neg f = 1$.

From now on, *SBCP*(a) refers to either *SYSTEM-BCP*(a) or *FORCED-SYSTEM-BCP*(a). The procedure *REVERSE-BCP*(g) described in this section is meant to find an "indirect reason" of the partial assignment g assigned to some variables of a system $\boldsymbol{S}$ by the procedure *SBCP*(a). The assignment g can be any subset of the set of elementary assignments contained in the list *DEDUCED* after running the procedure *SBCP*(a). Procedures of this kind are commonly used for conflict analysis in modern state-of-the-art SAT-solvers since the SAT-solver Grasp [12]. We need the procedure to prove some statements in the subsequent consideration. In Section 10, we will also use the procedure for conflict analysis.

Let a be a partial assignment to the pin variables of the system $\boldsymbol{S}$, and let an assignment g be deduced by *SBCP*(a). Let the procedure *REVERSE-BCP*(g) has the special goal, to deliver a partial assignment $a^{\complement}$ to pin variables of the system S such that $a^{\complement} \subseteq a$ and $a^{\complement}$ can be treated as a reason of g, i.e. g can be deduced under *SBCP*($a^{\complement}$).

The procedure *REVERSE-BCP*(g) takes as input the assignment g**,** as well as both lists *DEDUCED* and *REASONS* constructed during *SBCP*(a). At the beginning all elementary assignments from g are marked in *DEDUCED* and the resulting assignment $a^{\complement}$ is empty. Then the procedure passes through the list *DEDUCED* from its end to the beginning. At each step it considers the current element (elementary assignment) b_i of the list. If the assignment b_i is marked, the procedure considers the element corresponding b_i in the list *REASONS*. If this direct reason of b_i is the empty assignment $\varnothing$, then the elementary assignment b_i is added to the resulting assignment $a^{\complement}$. Otherwise, all assignments of this direct reason are marked in the list *DEDUCED*. After that the procedure starts the next step.

Example 9. Consider the lists *DEDUCED* and *REASONS* generated by the procedure *FORCED-SYSTEM-BCP*(a) in Example 8 where $a = \{b = 0, y^1 = 0\}$. We would like to find an indirect reason $a^{\complement} \subseteq a$ of the conflict in this case, namely a reason of the assignment $f = 1, f = 0$. The chronology of running the procedure *REVERSE-BCP*($f = 1, f = 0$) is represented in Fig. 9. The marks of the element of the list *DEDUCED* are denoted by the symbol *. At the beginning the elementary assignments $f = 1$, $f = 0$ are to be marked (Fig. 9a). When processing the assignment $f = 0$ and considering the list *REASONS* it is obvious that the direct reason of the assignment is $y^1 = 0$, $d = 1$. Both elementary assignments are marked in the list *DEDUCED* (Fig. 9b). When processing the assignment $f = 1$ we

have to mark the elementary assignment $b = 0$ that is the direct reason of $f = 1$ (Fig. 9b,c). Considering $d = 1$ provides no new marks because the direct reason $b = 0$ of $d = 1$ is already marked. After that, the

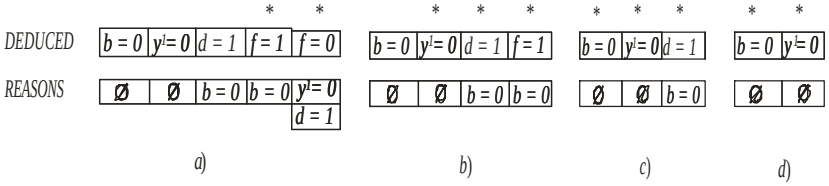


Fig. 9. *REVERSE-BCP*($f = 1, f = 0$)
a) before processing $f = 0$; *b*) before processing $f = 1$;
c) before processing $d = 1$; *d*) before processing $y^1 = 0$;

list contains two marked assignments $b = 0$ and $y^1 = 0$, that have the empty direct reason (Fig. 9d). Hence the assignment $a\phi = \{b = 0, y^1 = 0\}$ will be generated by the procedure as the indirect reason of the conflict. In this example, we have $a\phi = a$, however in general we will get $a\phi \subseteq a$. ⊗

The *REVERSE-BCP* procedure has the following useful property.

Theorem 22. Let $\boldsymbol{f^*(x,y,z)}$ be the extended permission function of a system $\boldsymbol{S}$, and let an assignment g be derived by the procedure $SBCP(a)$ for $\boldsymbol{S}$. Let the procedure *REVERSE-BCP*(g) provide the indirect reason b of the assignment g where $b \subseteq a$.

1. If g is an elementary assignment, then the clause representing the elementary implication $b \Rightarrow g$ is an implicate of the extended permission function $\boldsymbol{f^*(x,y,z)}$.
2. If the clause representing the assignment g is an implicate of the extended permission function $\boldsymbol{f^*(x,y,z)}$, then the clause representing the assignment b is also an implicate of $\boldsymbol{f^*(x,y,z)}$. ⊗

9.5 A System as a Block

By showing that a system $\boldsymbol{S}$ satisfies Axiom 3 through Axiom 6 (see Appendix 1), we prove that $\boldsymbol{S}$ can be considered as a normal block under both *SYSTEM-BCP* and *FORCED-SYSTEM-BCP*.

Theorem 23. A system of blocks is a normal block under both *SYSTEM-BCP* and *FORCED-SYSTEM BCP*. ⊗

When using the procedures *SYSTEM-BCP* and *FORCED-SYSTEM-BCP*, blocks of any complexity can be constructed. After finishing block construction, the question of how hierarchical SAT-solving should be organized can no longer be put off. This question is considered next.

10 Hierarchical SAT-Solving

In this paper, we try to avoid formulating unnecessary algorithms and heuristics, because we consider them to be an objective of fruitful research in the future. In this section, we firstly show that the top level organization of hierarchical SAT-solving in our framework is similar to CNF-oriented SAT-solving. This substantiates a "natural" transfer of well known ideas of that domain into our framework. So, we propose a brief overview of practical techniques and ideas efficiently applied for the best state-of-the-art CNF-based SAT-solvers. All these ideas can be efficiently used for organizing hierarchical SAT-solving. At the same time, hierarchical SAT-solving has some specific features that will be discussed.

10.1 Proving by Contradiction

Let y^d where $d \in \{0, 1\}$ denote a literal of the variable y, precisely, $y^1 = y$ and $y^0 = \neg y$. Then the unit clause y^d represents the assignment $y = \neg d$.

A system $\boldsymbol{S}$ with an output y tested to be constantly d ($d \in \{0,1\}$) is called *SAT-instance*. Let $\boldsymbol{S}$ implement a function $y(\boldsymbol{x})$ at the output y. An assignment $a \cup \{ y = \neg d\}$ is called a counterexample, if $y(a) = \neg d$. We will consider procedures of hierarchical SAT-solving which try to find a counterexample for $\boldsymbol{S}$ or to prove that a counterexample does not exist. These procedures assign y to $\neg d$ at a first step. If a procedure proves that there is no counterexample, then the system implements the constant d. Hence, the unit clause y^d is proven to be implicate of the permission function of the

system. Alternatively a procedure can prove that y^d is an implicate of the permission function of the system, then a counterexample does not exist.

10.2 Completeness of the system of axioms

Now we show that our proposed system of axioms is complete, i.e. there exists an algorithm which can correctly solve all possible SAT-instances in our theory.

Let an output y of a system $\boldsymbol{S}$ be tested to be constantly d. Consider the algorithm $\boldsymbol{A}$, running the procedure $SBCP(g)$ where $g = a \cup \{y = \neg d\}$ for all full assignments a to the input variables of $\boldsymbol{S}$. If there is an assignment g classified as not conflicting, then the algorithm $\boldsymbol{A}$ delivers g as counterexample. Otherwise, $\boldsymbol{A}$ reports that y is constantly equal to d.

Theorem 24. The algorithm $\boldsymbol{A}$ is correct. ⊗

10.3 Reducing Hierarchical SAT-Solving to Testing Satisfiability of a CNF

Now we consider a way in which hierarchical SAT-solving can be reduced to testing satisfiability of a CNF.

Given a system $\boldsymbol{S}$, the conjunction $\boldsymbol{C_S}$ of observable CNFs of all the system's blocks is called *structurally observable* CNF of $\boldsymbol{S}$.

Theorem 25. Let an output y of a system $\boldsymbol{S}$ be tested to be constantly equal to d, and let $\boldsymbol{C_S}$ be the structurally observable CNF of $\boldsymbol{S}$. Then the CNF $\boldsymbol{C_S} \wedge y^{\emptyset d}$ is unsatisfiable iff $y(\boldsymbol{x}) = d$ where $\boldsymbol{x}$ is the vector of the system's input variables. ⊗

According to Theorem 25 a way of solving a SAT-instance $\boldsymbol{S}$ is to construct its structurally observable CNF $\boldsymbol{C_S}$ and then check the CNF $\boldsymbol{C_S} \wedge y^{\emptyset d}$ for satisfiability. This way, methods developed for CNF satisfiability testing can be applied directly to hierarchical SAT-solving. This way is currently used at practice for systems consisting of blocks which correspond to gates implementing elementary Boolean functions. In this case, the observable CNFs are identical to the characteristic CNFs of the same blocks, and the CNF $\boldsymbol{C_S} \wedge y^{\emptyset d}$ (or $\boldsymbol{C_S}$) is called conventional CNF. However, if blocks are big or complex, extracting complete observable CNFs will lead to blowing up the size of $\boldsymbol{C_S}$.

At the same time it is known that resolution proofs of unsatisfiability very often only use a portion of clauses from $\boldsymbol{C_S}$, and this portion can be very small (up to a few per cent) especially in the case of instances formulated for property checking [49,50]. The basic idea of hierarchical SAT-solving is "to hide" the observable CNFs inside of block models in a compact form and then extract only their *necessary* clauses on demand. This idea can be naturally implemented, because block models are used for the same activity as clauses during CNF-oriented SAT-solving w.r.t. modern practical techniques [12, 13, 9, 35]. We discuss a way of extending these techniques in Section 10.5

10.4 Testing Satisfiability of a CNF as a Case of Hierarchical SAT-Solving

A clause can be considered as a block of a special type. Namely, a clause can be viewed as an (virtual) OR-gate whose output is assigned to the fixed constant value 1. So, a CNF can be viewed as a system consisting of blocks of the same specific type.

To put it more precisely, a CNF $\boldsymbol{C}$ can be considered as a two-level combinatorial circuit $\boldsymbol{S}$ that has OR-gates in its first level (some inputs of OR-gates can be inverted), and each OR-gate implements a clause of the CNF. On the second level, the circuit has an AND-gate realizing the conjunction of the CNF's clauses. The task of checking the CNF $\boldsymbol{C}$ for unsatisfiability is reduced to testing whether the circuit $\boldsymbol{S}$ implements the constant Boolean function 0 or not. According to Section 10.1 the output of the circuit is to be assigned to 1 at the first step of hierarchical SAT-solving. Assigning the value 1 to the output of AND-gate necessary implies assigning values 1 to all inputs of the gate. Thus, the two level circuit $\boldsymbol{S}$ is reduced to the set of OR-gates whose outputs are assigned to 1, where each OR-gate of this set corresponds to a clause of the CNF $\boldsymbol{C}$.

Let us consider for example the OR-gate corresponding to a clause $\neg a \vee b$. The conventional CNF for the gate is $(\neg a \vee b \vee \neg y) \wedge (a \vee y) \wedge (\neg b \vee y)$ where y is the output variable of the gate. Under the assignment $y = 1$ to the output of the gate its CNF is reduced to the same clause $\neg a \vee b$ which is implemented by the gate. Thus, hierarchical SAT-solving for the considered two-level combinatorial circuit $\boldsymbol{S}$ is reduced to testing satisfiability of the original CNF $\boldsymbol{C}$. Therefore, testing satisfiability of a CNF is a case of hierarchical SAT-solving.

10.5 Basic Ideas of CNF-Oriented SAT-Solving and Their Relation to Hierarchical SAT-Solving

From the point of view of CNF-oriented SAT-solving, a clause is an elementary object that can deliver useful information. A clause is used to produce an implication (when under a current assignment the clause becomes unit) or to recognize a conflict (when under a current assignment the clause becomes empty (or falsified)). Blocks are used for the same activity in our framework, because they produce implications and identify conflicts. This opens a way of extending basic ideas of CNF-oriented SAT-solving into the general framework of hierarchical SAT-solving. To organize a procedure of hierarchical SAT-solving one can use ideas and techniques listed below.

10.5.1 Search Tree Traversal

Most contemporary state-of-the-art SAT-solvers [8,9,12,34,35,36] are based on the classical DPPL algorithm [37]. Given a CNF $\boldsymbol{C}$, a DPLL-algorithm based SAT-solver looks for an assignment satisfying the CNF. Search is organized as a binary tree by branching on variables of the CNF $\boldsymbol{C}$. If no satisfying assignment is found after a complete examination of the search tree, then $\boldsymbol{C}$ is unsatisfiable. A hierarchical SAT-solving can be organized in the same way by branching on the variables of the system under consideration (recall that the output of the system is assigned according to Section 10.1).

10.5.2 Restarts

Modern SAT-solvers [8,9,35] efficiently use the idea of restarts that was proposed in [38]. A restart means that a SAT-solver abandons a current search tree and starts a new one. Back leaps proposed in [39] can be considered as variety of the restart idea. A back leap leads to erasing the current path of the search tree up to a node $\boldsymbol{n}$, and to continuing the search starting with $\boldsymbol{n}$. Intuitively, restarts give the search procedure a chance to withdraw from bad regions where it got stuck.

Restarts can be used for hierarchical SAT-solving as well. Moreover our theory sheds light on the nature of "bad regions". Suppose a model $\boldsymbol{M}$ of a block of a system $\boldsymbol{S}$ does not recognize a conflict assignment α to its pin variables. Given a current assignment γ to variables of the system $\boldsymbol{S}$ where $\alpha \subseteq \gamma$. Since a branching procedure cannot "feel" the conflict, it has to continue SAT-solving by making additional assignments to the variables of the system. The worst time loss can be exponential in number of variables of the system, if restarts are not used.

The reason for a bad region is that the model $\boldsymbol{M}$ has not maximal implicativity (the assignment α is not recognizable as conflicting). Note that a subsystem of a system can be viewed virtually as a block, resulting in the same effect of a "bad region". Based on an estimation of implicativity and revealing the bad regions indicated above, one can develop "clever" heuristics for restarts. That opens a domain for future research.

At each node of the search tree a CNF oriented SAT-solver performs the following three procedures: a) chooses the next variable to split on and selects the value, 0 or 1, to be first assigned to the chosen variable; b) runs the *Boolean Constraint Propagation* (BCP) procedure; c) performs conflict analysis and backtracking, if a conflict is encountered (during BCP run). All the three procedures can be performed in a similar way for hierarchical SAT-solving as it is discussed below.

10.5.3 Decision Strategies

Decision strategies of modern SAT-solvers are based on a phenomenon revealed by Chaff's team [8], namely, variables of recently deduced conflict clauses are most preferable to be branch on. BerkMin's [9] and Siege's [10] heuristics take this phenomenon into account more accurately. Decision heuristics should be closely related to classes of SAT-instances to be solved. Research in the field of hierarchical SAT-solving can inspire new heuristics. As example we refer to the paper [11] in which an idea of using signal correlation is proposed for decision making in circuit oriented SAT-solvers (meant for equivalence checking). One more useful idea is using topological levels of variables [40]. All these ideas can be used for hierarchical SAT-solving because of its ability to learn conflict clauses in a similar way as CNF-oriented SAT-solving do this as shown below. The idea of using a justification queue consisting of some gates implementing elementary Boolean functions [14] is inspired by automatic test pattern generation technique [47] and can reduce the number of conflicts [48]. This idea can be extended as well into the general case of blocks implementing complex Boolean functions.

10.5.4 BCP

One can use the procedure *SYSTEM-BCP* or *FORCED-SYSTEM-BCP* described in the previous section as a prototype of a BCP-procedure for a hierarchical SAT-solver.

An important feature of BCP for modern SAT-solvers is watching two literals per clause. Firstly, the idea was proposed in SATO [34], but under their implementation, watched literals were recalculated during backtracks. In Chaff's implementation of the idea [8], this drawback was eliminated, speeding up BCP.

A motivation of this technique is to reduce the "blank" processing of a clause. A processing is blank, if it provides no new implication or conflict. As the cost of blank processing for a block can be much higher, a similar idea is necessary for hierarchical SAT-solving. We propose to use *accessing functions* to get access to the BCP-procedure of a block. In the case of a clause, the accessing function is "at most two literals of a clause are unassigned", and a clause is processed only, if this is the case.

In general, an accessing function t for a model $\boldsymbol{M}$ of a block is defined on the set q of all possible partial assignments to the pin variables of the block as follows. Let q^+ be the subset of q that consists of all partial assignments recognizable by the model $\boldsymbol{M}$ to be conflicting or implying, and $q^- = q \setminus q^+$ be the set of the assignments not classified to be conflicting or implying by the model $\boldsymbol{M}$. An accessing function is any function that satisfies the following two conditions: 1) $t : q \to \{0,1\}$, 2) $t^{-1} \supseteq q^+$. In other words, the accessing function t specifies an over approximation of the set q^+. Given a partial assignment a to the pin variables of the block, if $t(a) = 1$, then model $\boldsymbol{M}$ can recognize the assignment to be conflicting or implying, if $t(a) = 0$, then the assignment is not conflicting or implying for the model $\boldsymbol{M}$, i.e. $a \in q^-$.

For practical use, an accessing function should be represented in a fast checkable form and should provide most close over approximation of the set q^+. We hope the idea of accessing functions can be implemented efficiently or even implicitly as it is done for clauses in modern SAT-solvers. In the simplest case, the accessing function is the constant 1, i.e. the access is always allowed.

10.5.5 Conflict Analysis and Non-Chronological Backtracking

A *decision level* of a node of a search tree is equal to the length of the path leading to the node from the tree's root. A level of an elementary assignment a (let a be $v = s$) is the decision level of the node in which v was assigned to s. Conflict analysis starts in a conflicting situation in which a variable exists that is to be assigned with two opposite values or an unsatisfied clause exists that has not any unassigned variable. A goal of conflict analysis is to find a reason of the conflict, i.e. an assignment under which the BCP-procedure leads to the conflict. In modern SAT-solvers a reason is constructed by running BCP in backward direction and the goal reason is the first assignment b that contains exactly one elementary assignment done on the current level of the decision tree [12, 13].

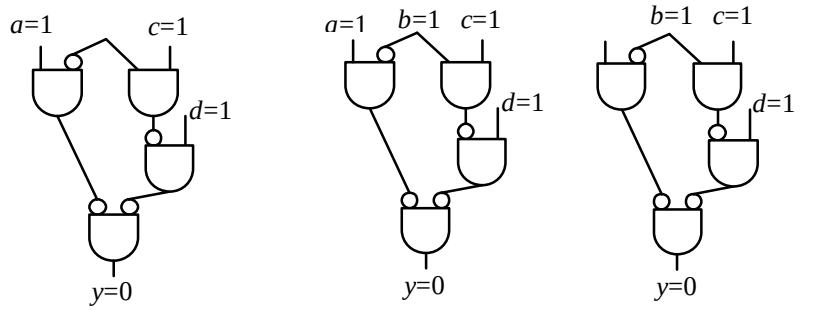
The goal assignment b of the same type can be constructed by the *REVERS-BCP-procedure* described in the previous section (after "tuning" the procedure on the goal). Effectively, this procedure behaves identical to the procedures used in modern SAT-solvers.

The constructed assignment b is used in two ways. Firstly, a clause $\boldsymbol{c}$ representing the assignment is added to the considered CNF, preventing a SAT-solver from visiting already investigated parts of the search space [12]. In hierarchical SAT-solving, the clause $\boldsymbol{c}$ also can be added to a database as (real or virtual) block-constraint having permission function $\boldsymbol{c}$. (It can be proven in the same way as for Theorem 22 that $\boldsymbol{c}$ is an implicate of the extended permission function of the system under consideration. Thus, such a block-constraint satisfies the fitting axiom.) The clause $\boldsymbol{c}$ is called *conflict* clause. Conflict clauses can be processed during hierarchical SAT-solving exactly as in CNF oriented SAT-solvers. At the same time a set of conflict clauses can be encapsulated into one bigger block-constraint (for example one can use BDDs for representation of sets of clauses [41] (BDD-based models are considered in Section 11.2)). A complex block-constraint can be processed during hierarchical SAT-solving as any other block.

Secondly, the assignment b is used to perform backtracking. In general, backtracking is *nonchronological* [12]. Let for example the current decision level be 100, and let the next maximal level of an elementary assignment contained in b be 50. Then one can perform backtracking to level 50 on which the conflict clause becomes a unit, and hence can be used by a BCP-procedure. A hierarchical SAT-solver can perform non-chronological backtracking in the same way.

10.5.6 Reason Reduction

Let an assignment a be a reason of a conflict, i.e. it leads to a conflict under a *BCP-procedure*. Suppose an assignment $a' \subset a$ is the reason of the same conflict. Then representing a' instead of a as the conflict clause is more beneficial, since the clause $\boldsymbol{c}'$ representing a' is a part of the clause $\boldsymbol{c}$ representing a. The idea of reason reduction is used in modern SAT-solvers when constructing conflict clauses (see, [53, 54] as example). It is desirable to have a procedure of this type inside of a block model. The following example demonstrates this concept.



a) a=1, *c*=1, *d*=1, *y*=0 (not conflicting/implying)
b) *a*=1, b=1, *c*=1, *d*=1, *y*=0 (conflicting)
c) *a*=1, *c*=1, *d*=1, *y*=0 (conflicting)

Fig.10 Reason reduction

Example 10. Suppose the circuit represented in Fig. 10 is a block model. The assignment $a = 1$, $c = 1$, $d = 1$, $y = 0$ is not conflicting or implying (Fig. 10a). After assigning $b = 1$ it becomes conflicting (Fig. 10b), however after removing $a = 1$ from it the resulting assignment is still conflicting (Fig. 10c). ⊗

We add an additional axiom to our system of axioms.

Axiom 7. A model $\boldsymbol{M}$ of a block $\boldsymbol{B}$ has the *REDUCE-REASON* procedure that satisfies the following conditions: 1) given a conflicting assignment a for the model, the procedure delivers a conflicting assignment a' for the model where $a' \subseteq a$; 2) given an elementary implication $a \Rightarrow b$ for the model, the procedure delivers an assignment a' such that $a' \subseteq a$ and $a' \Rightarrow b$ is an elementary implication for the model.

Note, that *REDUCE-REASON* can be trivial, providing no reason reductions, i.e. $a' = a$ for each a. So, axiom 7 can be considered as desirable however unnecessary. If a block is a system, then the procedure *REVERSE-BCP* can serve as the *REDUCE-REASON* procedure.

10.5.7 Data Base Management

Keeping all conflict clauses in a data base is expensive because, on the one hand, it can lead to blowing up the data base, and on the other hand, earlier conflict clauses can become irrelevant at the current stage of the search. Thus, clauses are removed from the data base depending on their size [8, 12], age, and activity in the conflict making process [9]. Furthermore, all unit conflict clauses and unit clauses of the initial CNF as well as literals set to 0 under the BCP-procedures triggered by these unit clauses are removed. We refer to this technique as *literal erasing*. All these ideas can be used in hierarchical SAT-solving. Besides, new ideas could be proposed relating to managing complex block-constraints after subsequent research.

10.6 Learning in Hierarchical SAT-Solving

Given a system $\boldsymbol{S}$, we understand learning as extending $\boldsymbol{S}$ with a block-constraint satisfying the fitting axiom (Section 9.2). As an example of learning consider Fig. 6. Learning is of fundamental importance due to the theorems 26, 27, and 28 considered in this section.

Theorem 26. Learning does not reduce the implicativity of a system $\boldsymbol{S}$. ⊗

Theorem 27. Given a system $\boldsymbol{S}$ not having maximal implicativity, there exists a block-constraint whose addition would make $\boldsymbol{S}$ a system with maximal implicativity. ⊗

Theorems 26 and 27 show that learning for a system cannot reduce its implicativity and always can increase implicativity up to the maximal possible. In practice, learning should aim to strongly increase the system's implicativity.

Now we discuss a difference between static and dynamic learning. By *dynamic* learning for a system $\boldsymbol{S}$ we mean on-the-fly learning during a SAT-solving process in which an output of the system

$\boldsymbol{S}$ is tested to be implementing a constant Boolean function. Dynamic learning results in the construction of specific block-constraints.

Let $\boldsymbol{B}¢$ be a block-constraint of a system $\boldsymbol{S}$ and $\boldsymbol{f}¢$ be the permission function of the block. Let y be an output of the system $\boldsymbol{S}$. We call the block $\boldsymbol{B}¢$ *restricted to the output y*, if $\boldsymbol{f}¢ = lit(y) \vee j(\boldsymbol{w})$ where $lit(y)$ is a literal of the variable y and $j(\boldsymbol{w})$ is a Boolean function depending on variables of the block $\boldsymbol{B}¢$ except of y. If lit(y) = y, $\boldsymbol{B}¢$ is *positively* restricted to the output y, otherwise it is *negatively* restricted to the output y.

Now suppose a SAT-instance is considered in which the output y is tested to be constantly 0. Then during SAT-solving y is assigned to 1 and all produced implicates of the extended permission function $\boldsymbol{f}^*$ of the system $\boldsymbol{S}$ have to be negatively restricted to y. For example, suppose a conflict clause $a \vee \neg b \vee c$ is deduced based on the technique outlined in the previous section. Then we can only affirm that $\neg y \vee a \vee \neg b \vee c$ is an implicate of the extended permission function $\boldsymbol{f}^*$, because using the literal erasing technique (Section 10.5.7) the literal $\neg y$ is to be erased due to initial assignment $y = 1$. Using the clause $a \vee \neg b \vee c$ during the SAT-solving instead of $\neg y \vee a \vee \neg b \vee c$ is correct while the SAT-solving is not finished.

Assume that dynamic learning was used only with the aim to increase implicativity of the system $\boldsymbol{S}$ being considered to be a "big" block of another system. In this case, a block-constraint $\boldsymbol{B}¢$ with the permission function $\neg y \vee a \vee \neg b \vee c$ can be added to the system $\boldsymbol{S}$.

In principle, it is possible to track whether a deduced clause really depends on the output variable y tested to be a constant. We leave this as subject of research for practical algorithms.

It is important to underline that, if the empty clause is deduced, then it depends on the tested output variable y, and $\neg y$ is real implicate of the permission function of the system under consideration (if y is tested to be the constant 0). Indeed, due to Theorem 23 the system is a normal block and due to Theorem 1 any implicate of the permission function of a normal block contains at least one output variable of the block. At the same time, adding a block-constraint with permission function $\neg y$ to the system, immediately increases implicativity of the system up to maximal possible due to Theorem 3 (if the system contains only output y), because $\neg y$ is the characteristic CNF of the permission function of the Boolean constant function $y = 0$. Thus, as soon as hierarchical SAT-solving is finished by proving that the considered system's output implements the tested constant, the system reaches maximal implicativity (when being restricted to that output). This observation can be formulated as the following theorem.

Theorem 27. Let a system $\boldsymbol{S}$ have one output only, and let hierarchical SAT-solving with dynamic learning be used to check whether $\boldsymbol{S}$ implements a given constant function. If there is no counter-example, hierarchical SAT-solving results in reaching maximal implicativity for $\boldsymbol{S}$. ⊗

By *static* learning we mean any learning technique except of dynamic learning. Static learning provides no restriction to permission functions of block-constraints except of the fitting axiom.

11 Cases of Hierarchical SAT-Solving

In the previous sections, we have shown that having compact block models with high implicativity is an essential prerequisite for hierarchical SAT-solving. In this section, we study different improved SAT-solving methods and show that they are either cases of hierarchical SAT-solving or that they can be fitted into our framework. In particular we consider circuit oriented SAT-solving, and combinations of SAT with binary decision diagrams (BDDs), which use block models with maximal implicativity.

11.1 Circuit-Oriented SAT-Solving

In circuit-oriented SAT-solving [11,15] combinatorial circuits consisting of gates implementing elementary Boolean functions are considered. To organize SAT-solving they use gate descriptions in the form of look up tables as in the equivalence checking domain [14]. We show that look up tables satisfy our system of axioms. Thus, current circuit-oriented SAT-solving can be viewed as a case of hierarchical SAT-solving in which the complexity of considered blocks is (very) small.

The circuit-oriented SAT-solvers [11,15] use a circuit representation based on AND and OR gate vertices, with INVERTERs either as separate vertices, or attributes on the gate inputs. The lookup table for 2-input AND is represented in Fig. 11. The idea of lookup tables is to support fast implication propagation. It is just what we need in our framework. Given a lookup table for a gate and an assignment a to the pin variables of the gate, the table classifies the assignment as conflicting, or

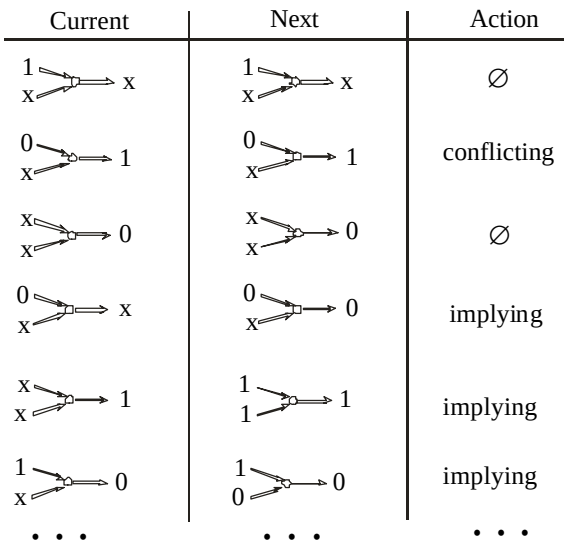


Fig.11. 2-input AND lookup table

implying an assignment, or not conflicting or implying. Thus, a lookup table specifies a *BCP-procedure* w.r.t. our terminology.

To prove that the lookup table for a gate is a model in our framework, one should check whether the *BCP-procedure* described by the table satisfies Axioms 4, 5, 6. Intuitively, it should be correct. The trivial way to prove it is to perform the exhaustive simulation on the lookup table under all possible partial assignments to the pin variables of the gate and to check directly for each assignment whether Axioms 4, 5, 6 are satisfied. The other way is to compare the result of the simulation with the result of the same procedure over a model of the gate. If the results are identical, then the models are observably coherent.

As an example we compare the 2-input AND lookup table represented in Fig. 11 with the CNF-based model considered in Example 2. The reactions of the models under the exhaustive simulation are represented in Fig. 12. Note that semantics of symbol "-" is the same as "x" in the Fig. 11 and Fig. 12, and denoting that the corresponding variable is unassigned. The symbol "c" denotes conflict. Since the models are observably coherent, the 2-input AND lookup table satisfies Axioms 4, 5, and 6. Moreover, since the considered CNF-based model has maximal implicativity (according to Example 3), then the 2-input AND lookup table has maximal implicativity.

One can check in the same way that lookup tables for the other types of gates under consideration are also models with maximal implicativity in our framework.

a	-	-	-	-	-	-	-	-	-	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
b	-	-	-	0	0	0	1	1	1	-	-	-	0	0	0	1	1	1	-	-	-	0	0	0	1	1	1
y	-	0	1	-	0	1	-	0	1	-	0	1	-	0	1	-	0	1	-	0	1	-	0	1	-	0	1

a) affecting assignments

a	-	-	1	-	-		-	0	1	0	0		0	0		0	0		1	1	1	1	1		1		1
b	-	-	1	0	0	c	1	1	1	-	-	c	0	0	c	1	1	c	-	0	1	0	0	c	1	c	1
y	-	0	1	0	0		-	0	1	0	0		0	0		0	0		-	0	1	0	0		1		1

b) reactions of the CNF-based model

a	-	-	1	-	-		-	0	1	0	0		0	0		0	0		1	1	1	1	1		1		1
b	-	-	1	0	0	c	1	1	1	-	-	c	0	0	c	1	1	c	-	0	1	0	0	c	1	c	1
y	-	0	1	0	0		-	0	1	0	0		0	0		0	0		-	0	1	0	0		1		1

c) reactions of the lookup table

Fig 12. Exhaustive simulation of AND gate $y = a \wedge b$ models

11.2 BDD Based Models

Reduced ordered binary decision diagrams [42] (BDDs) are commonly used as canonical representations of Boolean functions. Until now, BDDs are indispensable in the EDA domain. However, impressive recent advances in SAT resulted in SAT-tools undermining that position. At the same time, BDD models can be considered to be very good candidates for organizing hierarchical SAT-solving within our framework, as BDDs contain complete information about Boolean functions in a compact form. For implementing a *BDD-BCP-procedure* certain aspects of this information must be extracted quickly.

Initial attempts to combine BDDs and SAT-solvers were made by "topological division of spheres of influence", namely by describing the output or input part of the circuit under consideration using BDDs (see [18, 19] as examples). There, BDDs were intended for recognizing conflicts only. In a recent paper [16], the authors consider SAT-instances in the form of conjunctions of BDDs. After constructing BDDs for groups of clauses of a given CNF as preprocessing step, they use them during

SAT solving both for detecting conflicts and for producing implications. Their approach is very close to ours, as this procedure can be viewed as a *BDD-BCP* procedure for blocks represented by such BDDs. The difference is that in our framework, firstly, any mathematical model (not only BDDs) can be used as block model, and secondly, we follow a top-down approach in constructing models for typical design blocks. In contrast, they, in some sense, heuristically reverse engineer blocks in a bottom-up fashion (which can be done in our framework as well, when beneficial, such as for irregular control logic). Finally, we introduce the fundamental notion of implicativity, used for constructing the strict theory proposed above.

In this section, we consider reduced ordered BDDs with complemented edges [43]. This model is widely used in the EDA domain and provides a more compact representation for many Boolean functions than those of [16]. We outline the *BDD-BCP*-procedure and prove that the resulting BDD based block models have maximal implicativity.

11.2.1 BDD as a model

In the following we refer to reduced ordered BDDs with complemented edges simply as BDDs. A BDD, as shown in Fig. 13, is a directed acyclic graph with one source node denoting the represented function *f*, and one sink node (labeled 1). Each internal (not source or sink) node carries a variable symbol and has two outdoing edges labeled with 0 and 1, respectively. Dotted edges are complemented. To find the value of the Boolean function for a variable assignment a to the variables of *f*, the graph is traversed from the root to the sink node, while calculating the number of complemented edges on the path. If this number is odd, then $f(a) = 0$, otherwise $f(a) = 1$. For example, on the path *f*,*y*,0,*a*,1,*b*,0,*z*,1,*c*,0,1 there are two complemented edges: the second and the last. Hence, the value of *f* represented by this BDD is equal to 1 for the assignment $y = 0$, $a = 1$, $b = 0$, $z = 1$, $c = 0$.

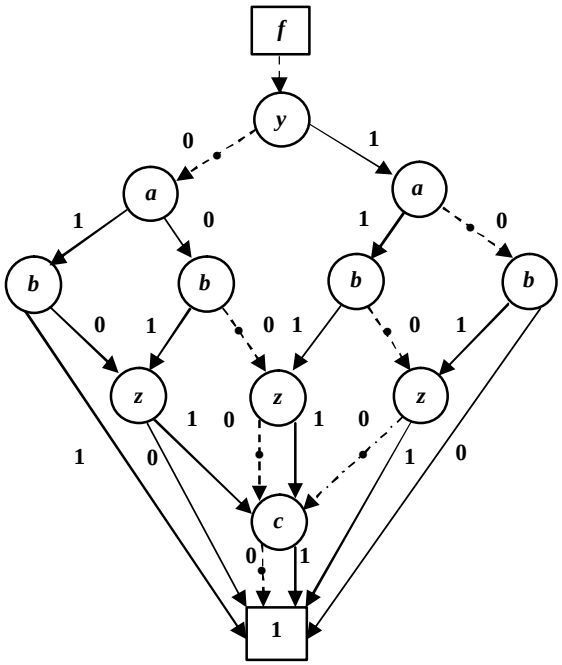


Fig 13. BDD for the permission function of 1-Bit-Adder

For a given block $\boldsymbol{B}$ implementing a vector function $\boldsymbol{y} = y(\boldsymbol{x})$, where $\boldsymbol{x} = (x_1,\ldots,x_n)$ and $\boldsymbol{y} = (y_1,\ldots,y_m)$ are the vectors of input and output variables, and $y = (y_1,\ldots,y_m)$, we use a BDD $\boldsymbol{D}$ of its permission function $\boldsymbol{f}$ as model. The permission function $\boldsymbol{f}$ can be calculated as conjunction of $y_1 = y_1(\boldsymbol{x}),\ldots, y_m = y_m(\boldsymbol{x})$. For example, the BDD of Fig. 13 represents the permission function of the two-output block of 1-Bit-Adder (Fig.1).

Let a be a partial assignment to the pin variables of the block $\boldsymbol{B}$. The *BCP-BDD*-procedure must correctly classify the assignments a to be conflicting or implying. It can be implemented on the basis of the following property of a BDD (Theorem 29) by considering a labeled graph $\boldsymbol{D}_a$, which is constructed from $\boldsymbol{D}$ as follows: For each elementary assignment from a, we remove from $\boldsymbol{D}$ all edges corresponding to the opposite value. (For example, if a contains $b = 1$, we mark all edges for $b = 0$ as removed). Let Cube(a) be the set of all complete assignments containing a.

Theorem 29. Given a BDD $\boldsymbol{D}$ and a partial assignment a to the variables of $\boldsymbol{D}$, each complete assignment of Cube(a) dissatisfies the function *f* represented by $\boldsymbol{D}$ *iff* any path leading from the source node *f* to the sink node 1 in $\boldsymbol{D}_a$ contains an odd number of complemented edges. ⊗

Let the *BDD-BCP*(a) procedure classify an assignment a:
a) To be conflicting, if any path leading from the source node *f* to the sink node 1 in $\boldsymbol{D}_a$ contains an odd number of complemented edges.
b) To imply an elementary assignment b, if a is not conflicting and any path leading from the source node *f* to the sink node 1 in $\boldsymbol{D}_{a \cup \neg b}$ contains an odd number of complemented edges.

Theorem 30. BDD is a consistent model with maximal implicativity and maximal strong implicativity under the *BDD-BCP*-procedure. ⊗

We see that the BDD model has nice properties: It is consistent, provides maximal implicativity and maximal strong implicativity, and hence is observably coherent to any other consistent model with maximal implicativity for the same block (due to Theorem 15).

11.2.2 Detailed *BDD-BCP*-Procedure

The *BDD-BCP*-procedure can be implemented in different ways. A possible way is to develop a procedure for testing whether an assignment a is conflicting, and, if a is not conflicting, to apply the procedure for all assignments $a \cup \neg b_i$ to check the implication $a \Rightarrow b_i$ (where $\neg b_i$ is an elementary assignment to a variable not contained in a).

In this section, we scratch a method that needs only three traversals of the graph under consideration to deduce all implications $a \Rightarrow b_i$. Consider a partial assignment a and graph $\boldsymbol{D}_a$. A level of a node v of the graph $\boldsymbol{D}_a$ is defined as the length of the longest path leading from the source node f to this node. Let an edge g connect nodes of levels i and j where $i < j - 1$, then g is called *passing through* each level k where $i < k < j$.

1. The *BDD-BCP*-procedure traverses the graph $\boldsymbol{D}_a$ in topological order (i.e. the source node is passed first (level 0), and nodes with smaller level (closer to the source) are passed before nodes of higher level). During the traversal, two Boolean values *t_odd* and *t_even* are recalculated for each ***node*** v. In the beginning, let *t_odd* = *t_even* = 0 for all the nodes. In the end of the traversal *t_odd* = 1 (*t_even* = 1) for a node v *iff* there is a path from the source node f to v that contains an odd (even) number of complemented edges. If the counter *t_even* for the sink node 1 is equal to 0 in the end of the traversal, then a is classified to be conflicting, and the procedure resets all the counters to 0 and stops.

2. The *BDD-BCP*-procedure traverses the graph $\boldsymbol{D}_a$ in reverse topological order. During the traversal, two Boolean values *r_odd* and *r_even* are recalculated for each ***edge*** g. In the beginning, let *r_odd* = *r_even* = 0 for all the edges. In the end we have *r_odd* = 1 (*r_even* = 1) for an edge g *iff* there is a path starting with g to the sink node 1 such that it contains an odd (even) number of complemented edges.

3. The procedure visits all the nodes of the graph $\boldsymbol{D}_a$, and for each node v and each outgoing edge g the procedure decides, based on *t_odd*, *t_even*, *r_odd*, *r_even*, whether there is a path from the source node of the graph to its sink passing through both node v and edge g, and has an even number of complemented edges. If there is no such a path for all the nodes corresponding to a variable z and for all the edges labeled with z = 0, as well as for all edges passing through the level of the variable z, then the assignment a is classified to be implying the elementary assignment $z = 1$. (Analogous, for implying $z = 0$). In the end of visiting a node v, *t_odd*, *t_even*, *r_odd*, *r_even* are to be reset to 0 of the node v and its outgoing edges.

11.3 CNF-Based Models for Tree-Like System

A system of normal blocks is called tree-like, if each of its variables feeds not more than one block input in the system, each block has one output, and the system is connected (each of its inputs is connected by a path with the output of the system).

In the recent papers [4,21-23], the author proposes a technique of constructing a CNF-based model for a group of gates. The technique is based on a description of a gate by a set of implications, and on substituting the implications of some gates into implications of other gates to eliminate the internal variables of the group. The proposed technique leads to impressive speed-ups for verification of microprocessors especially when being applied to tree-like groups of gates containing chains of multiplexers (if-then-else gates). We will refer to the technique as *gate merging*. In another recent paper [44] the authors report about successful use of their prepossessing engine NiVER that can be considered as a version of classical *resolution based* procedure VER [45]. In this section, we show that both techniques lead to the same CNF-based model when being applied to the same tree-like system $\boldsymbol{S}$ in which all blocks have CNF-based models with maximal implicativity. We also show that the same result can be obtained by using a procedure of *existential quantification*. The procedure constructs the permission function $\boldsymbol{f}$ of the system $\boldsymbol{S}$ as CNF by means of existential quantification of the extended permission function $\boldsymbol{f}^*$ of the system on the internal variables of $\boldsymbol{S}$. Finally, we show that the resulting CNF-based model generated for the system $\boldsymbol{S}$ by each of the three procedures has maximal implicativity.

11.3.1 Gate Merging

In this section, we consider the gate merging technique for constructing the CNF model of a tree-like system.

As first step, CNFs for gates are transformed into sets of implications, and we consider this transformation. Let C be a CNF depending on a variable y. Let C_y (respectively, $C_{Øy}$, $C_{\text{-}y}$) denote the CNF that consists of all clauses, where each of them is obtained from a clause of C containing the literal y (respectively, the literal $Øy$, or no literal of y) by removing the literal. For example, if $C = (a \vee \neg y) \wedge (b \vee \neg y) \wedge (\neg a \vee \neg b \vee y)$, then $C_y = (\neg a \vee \neg b)$, $C_{Øy} = a \wedge b$. However, $C_{\text{-}y} = 1$ because all clauses depend on the variable y in C. By the definitions above

$$C = (C_y \vee y) \wedge (C_{Øy} \vee Øy) \wedge C_{\text{-}y}. \qquad (3)$$

Consider a tree-like system consisting of two blocks $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$ (Fig. 14).

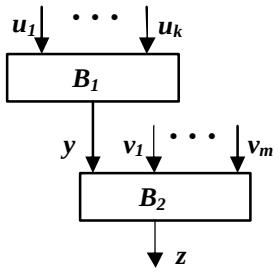


Fig. 14 A tree-like system of two blocks

Let the permission function of block $\boldsymbol{B_1}$ (block $\boldsymbol{B_2}$) be represented by CNF C^1 (CNF C^2 accordingly). Due to Theorem 1, all clauses of C^1 must contain the variable y that is the output variable of the block $\boldsymbol{B_1}$. Thus, $C^1_{\text{-}y} = 1$ and $C^1 = (C^1_y \vee y) \wedge (C^1_{Øy} \vee Øy)$. A clause containing literal y ($Øy$) is called *positive* (accordingly *negative*) *w. r. t. the variable y.* All positive clauses w. r. t. the variable y of the set C^1 can be obtained by performing the operation of logical addition (i.e. disjunction "$\vee$") of the CNF C^1_y and literal y. A clause c $\vee$ y which is positive w.r.t. the variable y can be represented as implication $\neg \boldsymbol{c} \Rightarrow y$ because $\boldsymbol{c} \vee y = \neg \boldsymbol{c} \Rightarrow y$. The negation of the clause $\neg \boldsymbol{c}$ can be rewritten as elementary conjunction (for example, $\neg (\neg a \vee \neg b) = a \wedge b$)). Hence, instead of considering the CNF C^1, a set of positive implications of the form *conjunction* $\Rightarrow y$ and a set of negative implications of the form *conjunction* $\Rightarrow \neg y$ can be considered. In a positive (respectively, negative) implication, *conjunction* is the negation of a clause from the set C^1_y (respectively, $C^1_{Øy}$). The CNF C^2 of the block $\boldsymbol{B_2}$ can also be represented as union of the sets of positive and negative implications w.r.t. the output z of $\boldsymbol{B_2}$.

Remind, x^e where $e \in \{0, 1\}$ denote a literal of the variable x, precisely, $x^1 = x$ and $x^0 = \neg x$. The method of *gate merging* being applied to the system depicted in Fig. 14 produces the implication *conjunction1* $\wedge$ *conjunction2* $\Rightarrow z^e$ for each pair of implications *conjunction1* $\Rightarrow y^d$ and *conjunction2* $\wedge$ $y^d \Rightarrow z^e$ in which the first implication is substituted into the second one (instead of the literal y^d). After that, the

$a \vee \neg y$ $b \vee \neg y$ $\neg a \vee \neg b \vee y$	$\neg a \Rightarrow \neg y$ $\neg b \Rightarrow \neg y$ $a \wedge b \Rightarrow y$	$\neg y \vee z$ $\neg c \vee z$ $y \vee c \vee \neg z$	$y \Rightarrow z$ $c \Rightarrow z$ $\neg y \wedge \neg c \Rightarrow \neg z$	$c \Rightarrow z$ $a \wedge b \Rightarrow z$ $\neg a \wedge \neg c \Rightarrow \neg z$ $\neg b \wedge \neg c \Rightarrow \neg z$	$\neg c \vee z$ $\neg a \vee \neg b \vee z$ $a \vee c \vee \neg z$ $b \vee c \vee \neg z$
a) CNF model for AND gate	b) implication model for AND gate	c) CNF model for OR gate	d) implication model for OR gate	e)implication model for the system	f) CNF model for the system

Fig. 15 Gate merging

system of blocks is replaced with one block having the set of produced implicates as model. The resulting set of implicates can be represented in a CNF form.

Example 11. Let the block $\boldsymbol{B_1}$ be a two-input AND-gate and the block $\boldsymbol{B_2}$ be the two-input OR-gate. The method of gate merging is illustrated by Fig. 15. ⊗

For tree-like systems containing more than two blocks, gate merging is performed for two-block tree-like subsystems until the system is reduced to one block. Thus, the method of [4] results in constructing a CNF-based model for a normal block which is equivalent to a tree-like system [4]. Moreover, we show in Section 11.3.4 that the constructed CNF is the characteristic CNF for the system considered as big block, if all system's blocks have characteristic CNFs as their models.

11.3.2 Resolution Based Method

Now we show that the same model as in the previous section can be constructed by applying the resolution based method of [44].

According to [45], to remove a variable y from a CNF C one can resolve each pair of clauses that can be resolved by the variable y (see Section 4 for the definition of resolution used in the paper). All resolvents are added to C and all clauses containing y are removed from C. The resulting CNF $C\ell$ is satisfiable *iff* the original CNF C is satisfiable. The resolution based method is illustrated by Fig. 16 for eliminating the variable y form the CNF that describes the system considered in Example 11. We see that the resulting CNF is the same as for gate merging.

Theorem 31. Let $\boldsymbol{S}$ be a tree-like system of two blocks $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$. Let the characteristic CNFs C^1 and C^2 of block $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$ be used as their models, respectively. Then the gate merging and the resolution based methods produce the same resulting CNF C. ⊗

$a \vee \neg y$ $b \vee \neg y$ $\neg a \vee \neg b \vee y$	$\neg y \vee z$ $\neg c \vee z$ $y \vee c \vee \neg z$	$\neg a \vee \neg b \vee y$, $\neg y \vee z$ $a \vee \neg y$, $y \vee c \vee \neg z$ $b \vee \neg y$, $y \vee c \vee \neg z$	$\neg a \vee \neg b \vee z$ $a \vee c \vee \neg z$ $b \vee c \vee \neg z$	$\neg a \vee \neg b \vee z$ $a \vee c \vee \neg z$ $b \vee c \vee \neg z$ $\neg c \vee z$
a) CNF model for AND gate	b) CNF model for OR gate	c) resolved pairs of clauses	d) resolvents	e) resulting CNF for the system

Fig. 16 Resolution based method

The resolution based method can be applied for elimination of the internal variables of a tree-like system. If all blocks of the system have the characteristic CNFs as models, the method is equivalent to gate merging of the system due to Theorem 31.

11.3.3 Existential Quantification of a Tree-Like System

In this section, we introduce two-block quantification procedure and show that it is equivalent to the procedures given in Sections 11.3.1 and 11.3.2.

Let $\boldsymbol{B}$ be a normal block, let y be the single output variable of $\boldsymbol{B}$, and let C^* be the characteristic CNF of the block. Then $C^*_{\text{-}y} = 1$, because by Theorem 1 any implicate of the permission function of the block must contain the output variable y.

Theorem 32. Let $\boldsymbol{S}$ be a system of two normal blocks $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$. Let $\boldsymbol{B_1}$ have one output y and let y feed block $\boldsymbol{B_2}$. Let C^1 and C^2 be the characteristic CNFs of block $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$ respectively, let $C^1 = (C^1_y \vee y) \wedge (C^1_{\varnothing y} \vee \varnothing y)$, and let $C^2 = (C^2_y \vee y) \wedge (C^2_{\varnothing y} \vee \varnothing y) \wedge C^2_{\text{-}y}$. Then the permission function $\boldsymbol{f}$ of the system $\boldsymbol{S}$ is equal to $(C^1_y \vee C^2_{\varnothing y}) \wedge (C^1_{\varnothing y} \vee C^2_y) \wedge C^2_{\text{-}y}$. ⊗

Given a tree-like system $\boldsymbol{S}$ consisting of *two blocks* $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$ in which the output y of $\boldsymbol{B_1}$ feeds an input of $\boldsymbol{B_2}$, we specify the procedure of existential quantification of the system $\boldsymbol{S}$. We call the procedure *two-block quantification*. Let C^1 and C^2 be the characteristic CNFs of blocks $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$ respectively. Due to Theorem 32 the permission function $\boldsymbol{f}$ of the system $\boldsymbol{S}$ is equal to $(C^1_y \vee C^2_{\varnothing y}) \wedge (C^1_{\varnothing y} \vee C^2_y) \wedge C^2_{\text{-}y}$. The procedure first constructs the permission function $\boldsymbol{f}$ in this form (i.e. as the result of existential quantification of the extended permission function of the system on the variable y due to proof of Theorem 32). After that, it performs logical addition (the operation "∨") over pairs of CNFs C^1_y and $C^2_{\varnothing y}$, and $C^1_{\varnothing y}$ and C^2_y, under which each pair is transformed into a CNF. The resulting CNFs are united with the CNF $C^2_{\text{-}y}$ into the final CNF denoted by $C(\boldsymbol{B_1}, \boldsymbol{B_2})$.

Theorem 33. Let $\boldsymbol{S}$ be a tree-like system of two blocks $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$. Let C^1 and C^2 be the characteristic CNFs of blocks $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$ respectively which are used as their models. Then the two-block quantification, and the resolution based method, as well as the gate merging produce the same CNF. ⊗

11.3.4 Maximal Implicativity of the Resulting CNF

Firstly, we show that $C(\boldsymbol{B_1}, \boldsymbol{B_2})$ is the characteristic CNF of $\boldsymbol{f}$. Thus, due to Theorems 33 and 3 all three procedures considered in Sections 11.3.1, 11.3.2, and 11.3.3 construct the same model with maximal implicativity for any two-block tree-like system in which each block has the characteristic CNF as the model. Finally, we show that the existential quantification procedure for any multi-block tree-like system produces the characteristic CNF of the system.

Theorem 34. Let ***S*** be a tree-like system consisting of *two blocks* $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$ in which the output y of $\boldsymbol{B_1}$ feeds an input of $\boldsymbol{B_2}$. Then the CNF $C(\boldsymbol{B_1}, \boldsymbol{B_2})$ is the characteristic CNF of the permission function $\boldsymbol{f}$ of ***S***. ⊗

Now we specify the *existential quantification procedure for a tree-like system* ***S***. Until there is a two-block tree-like subsystem $\boldsymbol{S}^{\wr} = \{\boldsymbol{B_1}, \boldsymbol{B_2}\}$ in the current system $\boldsymbol{S}^*$ (in the beginning of the procedure $\boldsymbol{S}^* = \boldsymbol{S}$), the procedure applies the two-block quantification procedure for $\boldsymbol{S}^{\wr}$ and replaces the subsystem $\boldsymbol{S}^{\wr}$ of $\boldsymbol{S}^*$ with the block having the CNF $C(\boldsymbol{B_1}, \boldsymbol{B_2})$ model. Due to Theorem 33 the procedure is equivalent to the gate merging and the resolution based method, if all the blocks of the system have the characteristic CNFs as models.

Theorem 35. The procedure of existential quantification of a tree-like system ***S*** constructs the characteristic CNF $\boldsymbol{C}^*$ for the permission function of the system. ⊗

Note, Theorem 35 holds for tree-like systems containing blocks of any complexity and not only gates implementing elementary Boolean functions.

11.4 Other Techniques

Now we consider briefly some other recent techniques that can be fitted into our framework of hierarchical SAT-solving providing block-models with maximal or increased implicativity.

11.4.1. Using State Machines for Representation of Boolean Functions

In the paper [25], the authors propose to use a special kind of state machine for representing Boolean functions (SMURF). An example of the model is given for $ite(a, b \wedge (c \oplus d), d \wedge (a \oplus b))$ in Fig. 17 [25]. SMURF is an acyclic Mealy machine where transitions produce value assignments to variables, i.e. implicates in our terminology. SMURF has a source node (state) that corresponds to the Boolean function $\boldsymbol{f}$ represented by the model and a sink node marked with 1 (the source is marked with $\boldsymbol{f}$) (Fig.17). Edges are marked with pairs of assignments denoting the input and output of the SMURF on a transition. In Fig. 17 one of the edges leaving the root node (top left) is marked with $\neg a$; d meaning that reading the elementary assignment $a = 0$ the SMURF produces the elementary assignment $d = 1$ on the transition to the state $b \oplus c$ corresponding to the cofactor $\boldsymbol{f}_{a=0,\, d=1}$. Note, that nodes corresponding to the same cofactors are merged. Thus, under the assignment $a = 1$ (implying $b = 1$, $c = 0$) and under the assignment $b = 1$ (implying $c = 0$) the model switches from the state $ite(a, b \wedge \neg c, b \oplus c)$ to the state 1.

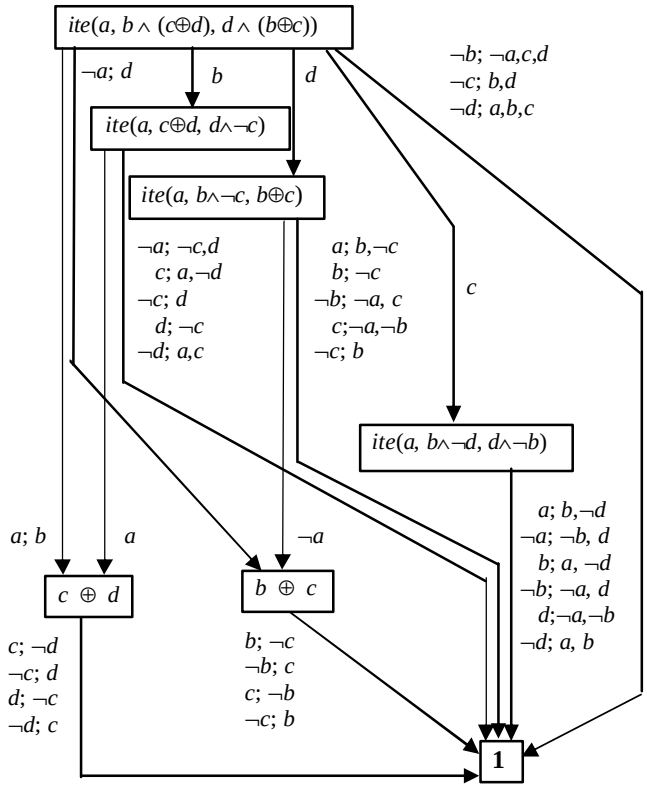


Fig. 17 SMURF for $ite(a, b \wedge (c \oplus d), d \wedge (a \oplus b))$

The method of constructing SMURFs is described in [25]. It can be shown that this model is consistent and provides maximal implicativity and maximal strong implicativity.

It is interesting to compare SMURFs with BDDs. The SMURF model implements a faster BCP than *BDD-BCP*. In the worst case, one has to pass along the longest path of a SMURF to extract relevant information from it. At the same time, one has to traverse a BDD three times completely (see Section 11.2.2). However, a SMURF can take much more space than the BDD for the same function. Firstly, the number of nodes of a SMURF is equal to the number of equivalence classes on the set of all possible cofactors of the Boolean function $\boldsymbol{f}$ under consideration (i.e. $O(3^n)$ in the worst case, where n is the number of variables of $\boldsymbol{f}$). At the same time, the number of nodes of the BDD for $\boldsymbol{f}$ is equal to the number of equivalence

classes on the set of cofactors that can be obtained by performing Shannon expansion of f under a given variable ordering (i.e. $O(2^n)$ in the worst case). Secondly, each node of a SMURF can have up to $2k$ outgoing edges where k is the number of variables a cofactor corresponding to the node depends on. BDD nodes have exactly 2 outgoing edges (except for the source and the sink). As a consequence of their bigger sizes SMURFs advantage in speed of BCP might be reduced due to the "cache architecture" of modern processors.

Our brief discussion illustrates how different mathematical models can be used according to the kind of block under consideration.

11.4.2. Using Pseudo-Boolean Constraints

In [24], the authors propose to use pseudo-Boolean constraints (PBC) as additional constraints to a SAT-solver and report about considerable speed-up for FPGA routing benchmarks. A PBC can be represented in the form

$$a_1y_1 + a_2y_2 + \ldots + a_ny_n \leq b \qquad (3)$$

where a_i, $b \in Z^+$ and y_i denotes either x_i or $\neg x_i$. The PBC in (3) corresponds to a threshold function which is unate (monotone) in each of its variables. PBCs can be used for implication deduction and fixing conflicts. For example, consider the constraint $a + b + c + d \leq 2$. If $a = b = 1$, then the only possibility to satisfy the constraint is to set $c = d = 0$. If $a = b = c = 1$, the constraint cannot be satisfied, thus we have conflict.

The authors also describe a procedure for deducing value assignments and fixing conflicts based on the PBC in (3). In our terminology this procedure corresponds to the BCP-procedure of a block-constraint with the permission function presented in (3). It can be shown that this model has maximal implicativity, maximal strong implicativity and is consistent.

11.4.3. Using Multiple Exclusive ORs

A multiple exclusive OR has the form

$$x_1 \oplus x_2 \oplus \ldots \oplus x_n \oplus d \qquad (4)$$

where x_i (i = 1,...,n) are Boolean variables and $d \in \{0,1\}$. An expression in (4) represents a linear Boolean function. The expression can be easily used to deduce implications and recognize conflicts. Consider the expression $a \oplus b \oplus c \oplus d$. Let $a = b = c = 1$, then $a \oplus b \oplus c = 1$. The only way to satisfy the expression under consideration is to set d to 0. However, if $a = b = c = d = 1$, we have a conflict. One can define a BCP-procedure for an expression in (4) based on counting the number of variable values assigned to 1 and checking whether the number is odd or even. It can be shown that such a BCP provides maximal implicativity and maximal strong implicativity and the model is consistent.

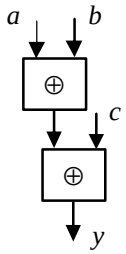


Fig.18 EX-OR chain

Consider the EX-OR chain depicted in Fig. 18. It implements the function $a \oplus b \oplus c = y$. Thus, the permission function of the chain can be represented as $a \oplus b \oplus c \leftrightarrow y$ or $a \oplus b \oplus c \oplus y \oplus 1$. Using multiple exclusive ORs for representation of EX-OR chains was proposed recently in [46].

11.4.4. Arithmetic Reasoning

Multipliers are known to be hard objects for SAT-solving. The main part of a multiplier is an addition network which calculates the sum of partial products. In a recent paper [26], the authors propose a technique called arithmetic reasoning that is based on column-wise calculation of the sum of partial products during SAT-solving. As a result "global" forward implications not delivered by *CNF-BCP* can be deduced. For example, multiplying 0 1 X 1 on 0 1 0 1 (where X denotes the undetermined value) by this technique delivers the value 0 for the two most significant product bits of a 4∗4-multiplier (Fig. 19). At the same time, it is possible that a circuit implementing the multiplier cannot produce the same values under BCP. To deduce an implication by the considered scheme it suffices to count only numbers of carry bits c_{ij} set to 1 and set to 0 in each column, not calculating the values of c_{ij} exactly [26]. Thus, the arithmetic reasoning procedure can be considered as a model of a block-constraint, which has pin variables a_{ij} (bits of addends) *and* b_i (bits of the product) as well as internal multi-valued variables for counting numbers of carries set to 1 or 0 in each column. The block

	8	7	6	5	4	3	2	1
1					$a_{14} = 0$	$a_{13} = 1$	$a_{12} = 0$	$a_{11} = 1$
X				$a_{24} = 0$	$a_{23} = X$	$a_{22} = 0$	$a_{21} = X$	
1			$a_{34} = 0$	$a_{33} = 1$	$a_{32} = 0$	$a_{31} = 1$		
0		$a_{44} = 0$	$a_{43} = 0$	$a_{42} = 0$	$q_{41} = 0$			
	$c_{17} = 0$	$c_{16} = 0$	$c_{15} = 0$	$c_{14} = 0$	$c_{13} = 1$	$c_{12} = 0$		
		$c_{26} = 0$	$c_{25} = 0$	$c_{24} = 0$	$c_{23} = 0$			
			$c_{35} = X$	$c_{34} = X$				
	$b_8 = 0$	$b_7 = 0$	$b_6 = X$	$b_5 = X$	$b_4 = X$	$b_3 = 0$	$b_2 = X$	$b_1 = 1$

Fig. 19 Arithmetic reasoning for 4*4-multiplier

a_{ij} – bits of addends, c_{ij} – carry bits, b_i – bits of the product

increases the implicativity of a system under consideration. However, it has not the maximal implicativity, at least because it does not provide backward implications.

11.4.5. Managing Don't Cares

In the recent paper [27], the authors propose a technique for using controllability and observability don't cares to improve performance of SAT-solvers. In this section, their technique of managing controllability is fitted into our framework.

A partial assignment a to the variables of a system $\boldsymbol{S}$ is called controllability don't care condition (CDC), if there is no full assignment b to the inputs of the system such that after running SYSTEM-BCP(b) the variables of the system take assignment g containing a. The technique [27] is meant for finding CDCs for a system given on the low (gate) level. CDCs revealed are represented by clauses which are added into the conventional CNF of the system.

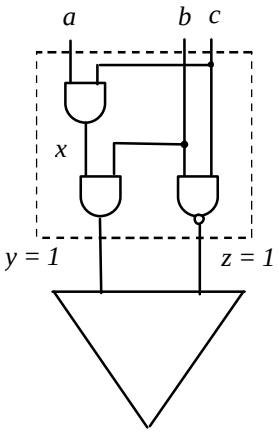


Fig.20 Controllability don't care

Preliminary in [27], a system is partitioned into fan-out free circuits referred to as cones, and CDCs are considered on inputs of cones. To find CDCs a subsystem feeding the variables under consideration is extracted. As CDCs may be time consuming to prove, the size of circuitry extracted is limited. Instead of extracting logic up to the primary inputs of the system they specify a number of cone levels to extract. In our framework, an extracted subsystem can be viewed as a virtual block having the conventional CNF model that is a part of the system CNF. By adding clauses representing CDCs into this part, the technique of [27] corresponds to increasing the implicativity of the block model.

Consider for example a part of a system (Fig. 20) in which a cone has inputs y and z. The assignment $a = \{y = 1, z = 1\}$ is a controllability don't care because no assignment to the inputs a, b, and c can cause a. Thus, the clause $\neg y \vee \neg z$ representing a is to be added to the conventional CNF of the system. On the other hand, the three-gate subsystem surrounded the dashed line in Fig. 20 can be considered as a block having the CNF $\boldsymbol{C} = (a \vee \neg x) \wedge (c \vee \neg x) \wedge (\neg a \vee \neg c \vee x) \wedge (x \vee \neg y) \wedge (b \vee \neg y) \wedge (\neg x \vee \neg b \vee y) \wedge (b \vee z) \wedge (c \vee z) \wedge (\neg b \vee \neg c \vee \neg z)$ as model. The clause $\neg y \vee \neg z$ is an implicate of $\boldsymbol{C}$ (because it is the resolvent of $x \vee \neg y$, $\neg x \vee c$, $\neg b \vee \neg c \vee \neg z$, $b \vee \neg y$). The CNF $\boldsymbol{C}$ does not provide the implication $z = 1 \Rightarrow y = 0$ under *CNF-BCP*, but after adding the clause $\neg y \vee \neg z$ into $\boldsymbol{C}$ the implication can be deduced. Thus, adding the clause $y \vee z$ increases implicativity of the block.

11.4.6. A Matrix Model

We have discussed a representative subset of models currently used in practical SAT-solving. For further progress in hierarchical SAT-solving, efficient models should be developed for typical blocks and structures of different application domains. Certainly, the models listed above could be considered as first candidates. However new specific models could be created. Developing SAT-models is an interesting topic of research, and the list of attractive models can be substantially extended. To illustrate this claim we consider a matrix model in this section.

In the simplest case of this model, information is stored in the form of a Boolean matrix $\boldsymbol{T}$ which represents the ON-set of a permission function under consideration. As example, consider the matrix $\boldsymbol{T}$ (Fig. 21,a) for the permission function $\boldsymbol{f}$ of 1-bit adder (Fig. 1,b).

A partial assignment to the pin variables of a block under consideration can be represented by the ternary vector. For example, the assignment $b = 1$, $z = 0$, $y = 0$ to the odered set of variables {*a*,*b*,*c*,*z*,y} can be represented by the vector – 1 – 0 0. Two ternary vectors of the same size are *orthogonal* by the *i*-th component, if they take opposite definite values in this component, i.e. one vector takes the values 0 and another takes the value 1. Thus, 0 0 0 0 0 and – 1 – 0 0 are orthogonal by the second component. Two ternary vectors are called orthogonal, if they are orthogonal by some component. A ternary vector is called orthogonal to a ternary (particularly, Boolean) matrix, if it is orthogonal to all its rows.

	a	*b*	*c*	*z*	*y*
	0	0	0	0	0
	0	0	1	1	0
	0	1	0	1	0
$\boldsymbol{T}$ =	0	1	1	0	1
	1	0	0	1	0
	1	0	1	0	1
	1	1	0	0	1
	1	1	1	1	1
$\boldsymbol{t}$ =	-	1	-	0	0

a) $\boldsymbol{t}$ is orthogonal to $\boldsymbol{T}$

	a	*b*	*c*	*z*	*y*
	0	**0**	**0**	**0**	**0**
	0	**0**	**1**	**1**	**0**
	0	**1**	**0**	**1**	**0**
$\boldsymbol{T}$ =	**0**	**1**	**1**	**0**	**1**
	1	**0**	**0**	**1**	**0**
	1	**0**	**1**	**0**	**1**
	1	**1**	0	0	1
	1	**1**	1	1	1
$\boldsymbol{t}$ =	1	1	-	-	-

b) removing elements

	c	*z*	*y*
$\boldsymbol{T}(\boldsymbol{t})$ =	0	0	1
	1	1	1

c) matrix $\boldsymbol{T}(\boldsymbol{t})$

Fig. 21 Matrix model for 1-bit adder

Given a matrix $\boldsymbol{T}$ representing the ON-set of a permission function $\boldsymbol{f}$, an assignment a represented by a ternary vector $\boldsymbol{t}$ can be correctly classified as conflicting iff $\boldsymbol{t}$ is orthogonal to $\boldsymbol{T}$ (because only in this case a can not be extended to an assignment from the ON-set of $\boldsymbol{f}$). In our example, $\boldsymbol{t}$ is orthogonal to $\boldsymbol{T}$ (Fig. 21.a). Thus, the assignment $b = 1$, $z = 0$, $y = 0$ is conflicting.

Now we consider conditions for recognizing implications. Let $\boldsymbol{t}$ = 1 1 - - -, as example. Let us remove from $\boldsymbol{T}$ all rows orthogonal to $\boldsymbol{t}$ and all columns in which the vector $\boldsymbol{t}$ has definite values (i.e. 1 or 0). The removed elements of $\boldsymbol{T}$ are shown in bold in Fig. 21,b. Let $\boldsymbol{T}(\boldsymbol{t})$ be the resulting matrix. As $\boldsymbol{T}(\boldsymbol{t})$ contains the unate column y consisting of ones (Fig. 21,c), the vector 1 1 - - 0 with the opposite value 0 for y is orthogonal to $\boldsymbol{T}$. Thus, the assignment $a = 1$, $b = 1$, $y = 0$ represented by this vector is conflicting. Hence, we can deduce the implication $a = 1, b = 1 \Rightarrow y = 1$.

Given a matrix $\boldsymbol{T}$ and the vector $\boldsymbol{t}$ representing an assignment a, by passing through $\boldsymbol{T}$ and performing bit-wise logical operations over $\boldsymbol{t}$ and rows of $\boldsymbol{T}$, it is possible to check whether a is conflicting or not and deduce all elementary implications $a \Rightarrow b_i$ (by identifying the unate columns in $\boldsymbol{T}(\boldsymbol{t})$). It can be show that such a model is consistent and has maximal implicativity and maximal strong implicativity. This model is faster than BDD, as it needs only one traversal of its data base (i.e. the matrix $\boldsymbol{T}$), and sometimes it can be more concise than BDD. As example, one can compare the BDD-model (Fig. 13) and the matrix model (Fig 21, a) for 1-bit adder. The BDD-model has 13 nodes and 23 edges, while the matrix model has 8 rows only.

An advanced version of this model is the representation of the ON-set $\boldsymbol{N^1}$ of a permission function $\boldsymbol{f}$ by a ternary matrix $\boldsymbol{T}$ in which each row represents a cube of $\boldsymbol{N^1}$. In this case, $\boldsymbol{T}$ is a representation of a disjunctive normal form (DNF) $\boldsymbol{D}$ of $\boldsymbol{f}$. A minimized DNF $\boldsymbol{D}$ for $\boldsymbol{f}$ can be found by using Boolean minimization procedures, for example ESPRESSO [51], BOOM [52].

12. Discussion and Further Research

12.1 Summary

In this paper, we have proposed a theoretical foundation for hierarchical SAT-solving. We have introduced 6 axioms, which a block must satisfy, as well as a fundamental notion of implicativity. Normal blocks and block-constraints are distinguished in the proposed theory. We have proven that testing whether a normal block's output implements a constant Boolean function is trivial, if the block has maximal implicativity. It has been also shown that constructing a consistent model for a normal block results in reaching maximal implicativity and maximal strong implicativity. We have proven that a system of blocks is a normal block thus providing a way of constructing blocks of any complexity. Basic procedures of hierarchical SAT-solving operating on blocks similarly to clauses during CNF-based satisfiability testing have been outlined. We have shown that these procedures lead to increased implicativity of a system by adding block-constraints, resulting in reaching maximal implicativity, if

the system implements the tested constant Boolean function. Basic methods for measuring and estimating implicativity have been also proposed.

The main conclusion of this theoretical work is that *hierarchical SAT-solving is reduced to increasing implicativity of a system*.

We have proven the relevance and the potential of our theory by identifying many new and promising research topics as cases of hierarchical SAT-solving. As for experimental confirmation of the usefulness of the theory we simply refer to recent papers of many researchers [4,11,15,16,18,19,20, 24,25,26,27] reporting substantial progress. Almost all of these techniques increase the level of abstraction of SAT-solving by constructing blocks with maximal implicativity.

12.2 Why Could Hierarchical SAT-solving Be an Interesting Topic of Research?

We have shown that the proposed theory can be considered as a generalization of existing experience in practical SAT-solving. At the same time, there is one more reason to attract attention of researchers to the topic of hierarchical SAT-solving. Based on the supposition that the computing facilities of humanity for enumerating Boolean functions are very restricted, we have to focus on some interesting classes of Boolean functions. This supposition is based on the following analysis:

Suppose that Moore's Law is continuing for the next 1000 years, i.e. every 2 years the performance of computers doubles. Currently, we have computers with a frequency of 4 GHZ, i.e. performing $4*10^9 < 2^{38}$ clock circles per second. Suppose that the performance of computers will increase due to increase of their clock frequency. Thus, we will have computers with $2^{38} * 2^{500} = 2^{538}$ clock circles per second in 1000 years (however, currently we even haven't a physical model supporting this fantastic performance). Further suppose that we already have A such computers working in parallel where A is the number of atoms in the universe. In other words, each atom of the universe is used as such a computer. Currently A is estimated as less than $10^{100} < 16^{100} = 2^{400}$. Let each computer consider one new function at each clock circle and let all computers together never consider the same function. Thus, our super universe-computer will be able to enumerate m functions during 1000 years (or less than in 2^{35} seconds) where

$$m < (2^{400}) * (2^{538}) * (2^{35}) = 2^{973} < 2^{1024} = 2^{2^{10}}$$

Note that m is less than the number of all Boolean functions in 10 variables, whereas we are faced with real life Boolean functions of n = 1000000 variables (and this n still grows). Dividing $2^{2^{10}}$ by $2^{2^{1000000}}$ results in a number 0.0……0d having at least 2^{249999} zeros succeeding the decimal point. Thus, m is "infinitely small" in comparison with the number of Boolean functions of 1000000 variables, and we have to focus on some specific classes of Boolean functions which could be interesting for humanity.

Our next supposition is that really interesting (for needs of humanity) functions come from real world systems. But how could realistic functions be classified to provide a way for developing methods for them specially? To answer this question a natural way is to go to the current sources of real world functions and work with them trying to understand their general features. It is clear that real world systems are hierarchical. Thus, one can try to take into account hierarchy of real world SAT-instances.

12.3 Current Limitations

In this paper, we advisedly introduce some limitations. First of all, we restrict the systems considered to only two levels of hierarchy: we have some blocks on the low level and a system of blocks on the high level. Secondly, the SAT-solving process is sequential. At the same time realistic systems can have many levels of hierarchy and their components can behave and interact (or communicate) concurrently. However we believe that a rational way of research is to develop a particular theory for the beginning and substantially exploit it in practice before considering a more general theory. Note, even in the proposed framework, one can consider complex realistic hierarchical systems, as a system of blocks is again a block in our theory (thus, one can use more and more complex blocks).

In our theory, blocks are used only for storing and extracting useful information and they can "discover" neither new (not stored) conflicts nor implications. As soon as block models with the ability to discover new conflicts or implications are used or a way of running block models concurrently is described, a more general situation is considered. Thus, a more general theory generalizing the one proposed here could be developed. In this sense, our theory can be viewed as a generalization of many advanced techniques proposed in practical SAT-solving.

12.4 Further Research

Our theory opens a rich domain for future research. Since hierarchical SAT-solving comes down to increasing implicativity of a system, one can try to reach this goal by increasing implicativity of its blocks at a preprocessing step. Until now, preprocessing was considered as a stage of SAT-solving for each particular SAT-instance. Our theory opens a new direction of research: *constructing block models*. Block models can be developed for typical blocks of the design and for typical or regular structures. These models can be reused for different SAT-instances.

A constructed block model should have three properties: 1) increased implicativity (compared to conventional CNF-based models); 2) fast BCP-procedure (to quickly extract information stored in the model); 3) be compact (for practical use). Ideally, implicativity should be maximal. Note that providing maximal implicativity for a block can be unnecessary for a particular instance, because SAT-solving can be successfully finished without using some implicates or conflicts stored in the block model. However, since it is impossible to predict what part of the stored information will be really used for a tested SAT-instance and because the model is to be used repeatedly for many instances, implicativity should be as high as possible. At the same time for big or complex blocks, even models with increased implicativity can be useful, because they will provide global implicates and conflicts which can be reproduced under SAT-solving after appropriate branching only. As example, we refer to arithmetic reasoning for multipliers [26] discussed in Section 11.4.4.

According to our theory, a way for increasing implicativity of a subsystem (considered as a big block) is to use learning techniques based on adding block-constraints to the subsystem. However, this increases the size of the subsystem. Another way is to develop a special compact model with a specific BCP. We show in the paper that quite different mathematical constructions can be used for block models (and they have been already proven to be efficient). After further research, new mathematical models can be involved into practical SAT-solving.

Our concept allows gradually increasing the level of abstraction of practical SAT-solvers based on advances of block model designers and makes it possible to use different improved techniques in cooperation. For example, in a system one can use one kind of model for multipliers, another kind of model for arbiters, different models for typical structures of control logic, or simultaneously, various techniques inside of one block model. Intuition leads us to postulate the third principle of hierarchical SAT-solving: *using different models for different kinds of blocks.* This principle is well adjusted with a tendency of modern commercial tools to use different techniques in cooperation.

Another important and more challenging direction of research is dynamic learning. Adding block-constraints is a way of increasing implicativity of a system under considerations. The problem is to find optimal (in time) strategies for increasing implicativity up to the maximum for instances of a given application domain. Hierarchical SAT-solving inherits an analogical problem from CNF-based satisfiability testing (that is the lowest level of hierarchical SAT-solving). Currently, the problem is tackled by heuristics. In the theory of hierarchical SAT solving, additional information can be used, such as structural properties of the system and the notion of implicativity.

In the formal verification domain, very often SAT instances globally consist of two subcircuits describing some designs compared. In this case, according to theoretical results [7] and modern experience, most efficient dynamic learning strategies should lead to constructing block-constraints relating both designs. For example, in equivalence checking of two similar combinatorial circuits, block-constraints should describe equivalence relations between internal variables of the circuits. In this case, when hiding variables inside of block models, some important constraints could be lost and SAT-solving could be complicated. Thus, an important topic of research related to developing efficient dynamic learning strategies is finding *a rational granularity of sizes of normal blocks used for solving practical SAT-instances*.

At the same time we would like to underline the flexibility of hierarchical SAT-solving. A normal block can be considered as a block-constraint. Thus, instead of replacing a combinatorial subcircuit with a normal block, one can use the latter as a block-constraint to support SAT-solving process.

The main contribution of this paper is the creation of a strict axiomatic theory covering many advanced techniques in practical SAT-solving and the introduction of the new important notion of implicativity which is shown to be the core notion of SAT-solving. We believe that our theory will attract the attention of researchers resulting in substantial progress in practical SAT-solving.

Acknowledgements

We whish to acknowledge the helpful suggestions and comments of Wolfgang Günther, Angela Matrosova, Sean Safarpour, Peter Warkentin, and Klaus Winkelmann. We also like to thank Rolf Drechsler and Bernd Steinbach for their support.

References

[1] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. Design Automation Conference, pp. 317-320, 1999*.

[2] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BBDs. In *Proc. of TACAS'99, pp. 193—207, 1999*.

[3] Solidify: Static Functional Verification for HDL Designers, http://www.averant.com, *2004*.

[4] M.N. Velev. Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC '04), pp. 310-315, 2004.*

[5] K. Winkelmann, H.-J. Trylus, D. Stoffel, G. Fey. Cost-Efficient Block Verification for a UMTS Up-Link Chip-Rate Coprocessor. In *Proc. European Design and Test Conference, pp. 162-167, 2004.*

[6] D. Jackson. An intermediate design language and its analysis. In *Proc. ACM SIGSOFT Foundations of Software Engineering, Orlando, Florida, pp. 121-130, 1998.*

[7] E. Goldberg and Ya. Novikov. How good can a resolution based SAT-solver be? *Lecture Notes in Computer Science. Publisher: Springer-Verlag ,Vol. 2919 Theory and Applications of Satisfiability Testing: 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected and Revised Papers, Editors: Enrico Giunchiglia, Armando Tacchella, 2004, pp. 37-52*.

[8] M. Moskewicz, C. Madrigan, Y. Zhao, L. Zhang, and S. Malik, Chaff: Engineering an efficient SAT solver. In *Proc. ACM/IEEE Design Automation Conference, pp. 530 - 535, 2001*.

[9] E. Goldberg, Y. Novikov. BerkMin: A fast and robust SAT-Solver. In *Proc. European Design and Test Conference, pp. 142-149, 2002.*

[10] L. Ryan. Efficient Algorithms for Clause-Learning SAT Solvers. http://www.satlive.org/index.jsp, *2004*.

[11] F. Lu, L.-C. Wang, K.-T. Cheng, and R. Huang. A circuit SAT solver with signal correlation guided learning. In *Proc. European Design and Test Conference 2003*. http://cadlab.ece.ucsb.edu/downloads/UCSB_Circuit_SAT/09d_3_496.pdf

[12] J.P. Marques-Silva and K.A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers, vol. 48, pp. 506-521, 1999*.

[13] L. Zhang, C. Madgigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proc. Intl. Conf. On Computer-Aided Design, pp. 279 – 285, 2001*.

[14] A. Kühlmann, M. Ganai, and V. Paruthi. Circuit-based Boolean Reasoning. In *Proc. ACM/IEEE Design Automation Conference, pp. 232-237, 2001*.

[15] M.K. Ganai, L. Zhang, P. Ashar, A. Gupta, and S. Malik. Combining strengths of circuit-based and CNF-based algorithms for a high-performance SAT solver. In *Proc. ACM/IEEE Design Automation Conference, pp. 747-750, 2002*.

[16] R. Damiano and J. Kukula. Checking satisfiability of a conjunction of BDDs. In. *Proc. ACM/IEEE Design Automation Conference, pp. 818-823, 2003.*

[17] J.R. Burch and V. Singhal. Tight integration of combinatorial verification methods. In. *Proc. Intl. Conf. On Computer-Aided Design, pp. 570-576, 1998.*

[18] A. Gupta and P. Ashar. Integrating a Boolean satisfiability checker and BDDs for combinatorial equivalence checking. In. *Proc. Int'l Conf. on VLSI Design, pp. 222-225, 1997*.

[19] S. Reda and A. Salem. Combinatorial equivalence checking using Boolean satisfiability and binary decision diagrams. In *Proc. Design Automation and Test in Europe, pp. 122-126, 2001*.

[20] A. Gupta, M. Ganai, C. Wang, Z. Yang, and P. Ashar. Learning from BDDs in SAT-based Bounded Model Checking. In. *Proc. ACM/IEEE Design Automation Conference, pp. 824-829, 2003*.

[21] M.N. Velev. Exploiting Signal Unobservability for Efficient Translation to CNF in Formal Verification of Microprocessors. In. *Proc. Design, Automation and Test in Europe, pp. 266-271, 2004*.

[22] M.N. Velev. Using Positive Equality to Prove Liveness for Pipelined Microprocessors. In *Proc. Asia and South Pacific Design Automation Conference, pp. 316-321, 2004*.

[23] M.N. Velev. Using Automatic Case Splits and Efficient CNF Translation to Guide a SAT-Solver When Formally Verifying Out-of-Order Processors. In *Proc. Artificial Intelligence and Mathematics, pp. 242-254, 2004.*
[24] F.A. Aloul, A. Ramani, I.L. Markov, K.A. Sakallah. Generic ILP versus Specialized 0-1 ILP: An Update. In. *Proc. Intl. Conf. On Computer-Aided Design, pp. 450-457, 2002.*
[25] Franco et al. Function-Complete Lookahead in Support of Efficient SAT Search Heuristics. *Journal of Universal Computer Science* (to appear).
[26] M. Wedler, D. Stoffel, and W. Kunz. Arithmetic reasoning in DPLL-based SAT solving. In. *Proc. Design, Automation and Test in Europe, pp. 30-35, 2004*.
[27] S. Safarpour, A. Veneris, R. Drechsler, J. Lee. Managing Don't Cares in Boolean Satisfiability. In *Proc. Design, Automation and Test in Europe, pp. 260-265, 2004*.
[28] G.C. Tseitin. On the Complexity of Derivation in Propositional Calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part 2, 1968, pp. 115-125. Reprinted in J. Siekmann, and G. Wrightson, eds., Automation of Reasoning, Vol.2, Springer-Verlag, pp. 466-483, 1983*.
[29] A. Blake. Canonical Expression in Boolean Algebra. Dissertation, Chicago, *1937*.
[30] P.S. Poretski. On the method of solving logical equations and on the inverse method for mathematical logic (in Russian). *Sobranie protokolov zasedanit fis. Mat. Kasan 2, pp. 161-330, 1884*.
[31] O. Coudert, J.C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean functions. In. *Proc. Design Automation Conference, pp. 36-39, 1992*.
[32] O. Coudert, J.C. Madre. Fault Tree Analysis: 10^{20} Prime Implications and Beyond. *In Proc. Annual Reliability and Maintainability Symp, pp.240-245, 1993.*
[33] A. Biere. Limmat Satisfiability Solver. http://www2.inf.ethz.ch/personal/biere/projects/limmat/, *2004*.
[34] H. Zhang. SATO: An Efficient Propositional Prover. In *Proc. of International conference on Automated Deduction, Vol 1249, LNAI, pp. 272-275, 1997.*
[35] L. Ryan. Siege SAT Solver v.4 http.//www.cs.sfu.ca/~loryan/personal/, *2004.*
[36] R.J.Bayardo, R.C.Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In. *Proc. of 14th National Conference on Artificial Intelligence, pp. 203-208, 1997.*
[37] M. Davis, G. Longemann, D. Loveland. A Machine program for theorem proving. *Communications of the ACM. Vol 5, pp. 394-397, 1962*.
[38] C. P. Gomes, B. Selman, H. Kautz. Boosting combinatorial search through randomization. In *Proc. of the Fifteenth National Conference on Artificial Intelligence (AAAI'98), pp. 431-437, 1998.*
[39] S. Pilarski and G. Hu. SAT with Partial Clauses and Back-Leaps. In *Proc. ACM/IEEE Design Automation Conference, pp. 743-746, 2002*.
[40] E. Goldberg, Y. Novikov. Equivalence Checking of Dissimilar Circuits. In. *Proc. International Workshop on Logic and Synthesis. Laguna Beach, California, USA, pp. 244-251, 2003.*
[41] F. Aloul, M. Mneimneh, and K. Sakallah. Search-Based SAT Using Zero-Suppressed BDDs. In *Proc. Design, Automation, and Test in Europe, pp. 1082, 2002*.
[42] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp., 35 (8), pp. 677-691, 1986.*
[43] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Proc. Design Automation Conf, pp. 40-45, 1990*.
[44] S. Subbarayan, D.K. Pradhan. NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT instances. In *Proc. of Seventh International Conference on Theory and Applications of Satisfiability Testing, 2004.* http://www.satlive.org/index.jsp
[45] M. Davis, H. Putnam. A Computing procedure for quantification theory. *Journal of the ACM, v.7 n.3, p.201-215, July, 1960*.
[46] J.A. Roy, I.L. Markov, V. Bertacco. Restoring Circuit Structure from SAT Instances. In. *Proc. International Workshop on Logic and Synthesis. Temecula Greek CA. June, 2004.* http://www.eecs.umich.edu/~imarkov/pubs/misc/iwls04-sat2circ.pdf
[47] M.Abramovici, M.A.Breuer, and A.D.Friedman. *Digital Systems Testing and Testable Design, W.H.Freeman, 1990*.
[48] F. Lu, L.-C. Wang, K.-T. Cheng, J. Moondanos, and Z. Hanna. A Signal Correlation Guided ATPG Solver and its Application for Solving Difficult Industrial Cases. In . *Proc. ACM/IEEE Design Automation Conference, pp. 436-441, 2003*.
[49] E. Goldberg, Ya. Novikov. Verification of proofs of Unsatisfiability for CNF Formulars. In. *Proc. Design, Automation, and Test in Europe, 2003.*

[50] L. Zhang, S. Malik. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In. *Proc. Design, Automation, and Test in Europe, 2003.*
[51] R.K. Brayton et al. Logic Minimization Algorithms for VLSI Synthesis. *Boston, MA, Kluwer Academic Publishers, 1984, 192 pp.*
[52] J. Hlavicka, P. Fiser. BOOM – A Heuristic Boolean Minimizer. *Computers and Information, Vol. 22, No. 1, pp. 19-51, 2003.*
[53] N. Sörensson, N. Een. MiniSat v1.13 A SAT Solver with Conflict-Clause Minimization. http://www.lri.fr/~simon/contest/results/descriptions/solvers/minisat_static.pdf
[54] R. Gershman. HaifaSat – a new robust SAT solver. http://www.lri.fr/~simon/contest/results/descriptions/solvers/HaifaSat.pdf.

Appendix 1

Proof of Lemma 1: Follows from the definition of the permission function. ⊗

Proof of Theorem 1: First, note that the permission function of a normal block cannot be idle because it must determine values of block output variables. Suppose the theorem does not hold, i.e. a nonempty implicate $\boldsymbol{c}$ exists that does not contain any output variable. Let a be a full value assignment a to the input variables of the block such that $\boldsymbol{c}(a) = 0$. According to Lemma 1 there exist a pattern (a, b) that satisfies $\boldsymbol{f}$. On the other hand, since $\boldsymbol{c}$ is an implicate and $\boldsymbol{c}(a, b) = 0$, $\boldsymbol{f}(a, b) = 0$. ⊗

Lemma 2. *CNF-BCP* provides monotone classification of partial assignments.

Proof: Follows from the "monotone nature" of the procedure. Let *CNF-BCP* classify an assignment a as conflicting. Assigning some additional variables can only extend the total set of unit clauses derived by the procedure or provoke an earlier conflict. Thus, any assignment $a \cup g$ is classified to be conflicting. Analogically, extending an implying assignment a can lead to a conflict or the same elementary implications (and may be some additional ones) as for a.

Now we show that for any elementary implication $a \Rightarrow b_i$ the assignment $a \cup b_i$ is not conflicting. Let $a \Rightarrow b$ and b_i Î b. According to the procedure if $a \Rightarrow b$, then $a \cup b$ is not conflicting (as the procedure is finished in a situation when the variables are assigned to $a \cup b$ and there is no conflict). Then $a \cup b_i$ can not be conflicting, as otherwise $a \cup b$ must be conflicting due to proven above monotone classification of conflicting assignments. ⊗

Lemma 3. Let a Boolean function $\boldsymbol{f}(\boldsymbol{x})$ take value 1 for only one full value assignment a to its arguments, precisely $a = \{x_1 = d_1,\ldots,x_n = d_n\}$. Then any CNF C representing $\boldsymbol{f}$ must contain for each argument x_i a unit clause u_i that is satisfied by the elementary assignment $x_i = d_i$ from a.

Proof: CNF C representing the $\boldsymbol{f}(\boldsymbol{x})$ must contain at least one clause. Let $x_i = \neg d_i$ then C must take value 0. It can be done only, if C contains the unit clause u_i described in the lemma. ⊗

Lemma 4. Let $C(\boldsymbol{x},\boldsymbol{y})$ be a CNF representing the permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ of a normal block $\boldsymbol{B}$ and a be a full value assignment to inputs of the block. Then the *CNF-BCP* procedure provides a correct assignment b to outputs of the block under a, i.e. $b = \mathrm{BCP}(a)$ and $b = Y(a)$.

Proof: Make the assignment a to all input variables of the block. Since there is no clause containing input variables only (according to Theorem 1), there is no clause equal to 0 under a in $C(\boldsymbol{x},\boldsymbol{y})$.

Remove all satisfied clauses and literals. The resulting CNF C^* must implement a Boolean function that takes value 1 for only one full value assignment to its arguments. Indeed, due to Lemma 1 there is exactly one pattern (a, b) that satisfies $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$. Hence, CNF C^* takes value 1 for the pattern b only. According to Lemma 3, CNF C^* contains unit clauses from which *CNF-BCP* must deduce the full assignment b to the outputs of the block. ⊗

Let a be a partial value assignment to Boolean variables from a vector $\boldsymbol{z}$, we define $Cube(a)$ as the set of all possible full assignments b to variables from $\boldsymbol{z}$, where $a \subseteq b$.

Lemma 5. Let $C(\boldsymbol{x},\boldsymbol{y})$ be a CNF representing the permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ of a block $\boldsymbol{B}$, and let a be an assignment classified by $CNF\text{-}BCP(a)$ as conflicting for $C(\boldsymbol{x},\boldsymbol{y})$. Then the clause representing a is an implicate of the permission function $\boldsymbol{f}$.

Proof: *CNF-BCP*(a) procedure classifies a as conflicting assignment, if the *Cube*(a) of the Boolean space of variables ($\boldsymbol{x},\boldsymbol{y}$) doesn't contain a pattern satisfying the CNF $C(\boldsymbol{x},\boldsymbol{y})$. ⊗

Lemma 6. Let $C(\boldsymbol{x},\boldsymbol{y})$ be a CNF representing the permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ of a block $\boldsymbol{B}$, and let a imply b under *CNF-BCP*(a) procedure for $C(\boldsymbol{x},\boldsymbol{y})$. Then for each elementary assignment $b_i \in b$ the clause representing the elementary implication $a \Rightarrow b_i$ is an implicate of the permission function $\boldsymbol{f}$.

Proof: *CNF-BCP*(a) procedure classifies a as implying b_i, if the *Cube*($a,\neg b_i$) of the Boolean space of variables ($\boldsymbol{x},\boldsymbol{y}$) doesn't contain a pattern satisfying the CNF $C(\boldsymbol{x},\boldsymbol{y})$. ⊗

Proof of Theorem 2: Follows from Lemmas 2,4,5,6. ⊗

Proof of Theorem 3: According to Theorem 2 the CNF $\boldsymbol{C}^{\cdot}$ is a model under *CNF-BCP*.

Let $\boldsymbol{M}$ be an arbitrary model of $\boldsymbol{B}$ and a be an assignment recognizable by $\boldsymbol{M}$ as conflicting. Since a is a conflicting assignment, the clause $\boldsymbol{c}$ representing a is an implicate of the permission function $\boldsymbol{f}$. Consider any prime implicate $\boldsymbol{c}^{\complement}$ of $\boldsymbol{f}$ that is a part of $\boldsymbol{c}$. As far as $\boldsymbol{c}^{\complement}(a) = 0$ *CNF-BCP*(a) classifies the assignment a as conflicting for $\boldsymbol{C}^{\cdot}$.

Now, let $\boldsymbol{M}$ produce an implication $a \Rightarrow b$. Then for any elementary implication $a \Rightarrow b_i$ of $a \Rightarrow b$, the clause $\boldsymbol{c}$ representing the elementary implication is an implicate of the permission function $\boldsymbol{f}$. Consider any prime implicate $\boldsymbol{c}^{\complement}$ of $\boldsymbol{f}$ that is a part of $\boldsymbol{c}$. If $\boldsymbol{c}^{\complement}$ does not contain a variable assigned in b_i, then $\boldsymbol{c}^{\complement}(a) = 0$ and *CNF-BCP*(a) classifies the assignment a to be conflicting for $\boldsymbol{C}^{\cdot}$. If $\boldsymbol{c}^{\complement}$ contains a variable assigned in b_i, then under assignment a the clause $\boldsymbol{c}^{\complement}$ becomes a unit satisfied by b_i. Consequently, *CNF-BCP*(a) will deduce b_i, and a implies b_i in $\boldsymbol{C}^{\cdot}$.

So for any model $\boldsymbol{M}$ and any assignment a to pin variables we have: If a is classified as conflicting or implying by the model, it is also classified as conflicting or implying for the CNF $\boldsymbol{C}^{\cdot}$. Hence, the characteristic CNF $\boldsymbol{C}^{\cdot}$ of the permission function $\boldsymbol{f}$ has maximal implicativity. ⊗

Proof of Theorem 4: Consider an arbitrary partial value assignment a to the pins of the block B, such that *CNF-BCP*(a) classifies a to be conflicting or implying for the CNF $\boldsymbol{C}'$. The clause $\boldsymbol{c}^{\complement}$ can have an influence during the *CNF-BCP*(a) procedure in two cases:

1. Under a current assignment g, the clause $\boldsymbol{c}^{\complement}$ becomes empty in CNF $\boldsymbol{C}'$, and the procedure reports a conflict under a. In this case, the clause $\boldsymbol{c}$ is empty under the same assignment. Hence, if $\boldsymbol{c}^{\complement}$ is removed from $\boldsymbol{C}'$, the procedure also classifies a to be conflicting.
2. Under a current assignment g, the clause $\boldsymbol{c}^{\complement}$ becomes a unit u. If the unit u is not a part of the clause $\boldsymbol{c}$, the clause $\boldsymbol{c}$ is empty under g, and the procedure must report a conflict on $\boldsymbol{c}$ in both cases whether $\boldsymbol{c}^{\complement}$ is removed or not. If the unit $\boldsymbol{u}$ is a part of the clause $\boldsymbol{c}$, then under the current assignment g the clause $\boldsymbol{c}$ becomes the same unit u, and the procedure must deduce a value (from the unit $\boldsymbol{u}$) in both cases, no matter whether $\boldsymbol{c}^{\complement}$ is removed or not. ⊗

Proof of Theorem 5: Consider the CNF $\boldsymbol{C}^{*}$ obtained by the procedure of exhaustive simulation of $\boldsymbol{M}$ (just before removing any clauses). The CNF $\boldsymbol{C}^{*}$ must simulate under *CNF-BCP* the same vector function $\boldsymbol{y} = Y(\boldsymbol{x})$ as $\boldsymbol{M}$ does. Indeed, for each full value assignment a to the inputs $\boldsymbol{x}$ the model $\boldsymbol{M}$ produces a full assignment b to outputs $\boldsymbol{y}$ where $b = Y(a)$. At the same time, for each elementary assignment $b_i \in b$ the CNF $\boldsymbol{C}^{*}$ contains the clause representing elementary implication $a \Rightarrow b_i$. Hence, the $\boldsymbol{C}^{*}$ simulates $\boldsymbol{y} = Y(\boldsymbol{x})$ under *CNF-BCP*.

By construction, $\boldsymbol{C}^{*}$ observably covers $\boldsymbol{M}$: Each conflicting assignment a is represented by a clause in $\boldsymbol{C}^{*}$. Thus, *CNF-BCP* classifies a as conflicting for $\boldsymbol{C}^{*}$. Each elementary implication $a \Rightarrow b_i$ is represented by a clause in $\boldsymbol{C}^{*}$. Thus, *CNF-BCP* can deduce the same implication or fix conflict as a result of unit clause propagation in $\boldsymbol{C}^{*}$.

Since removing covering clauses from a CNF one after another keeps obtained CNFs observably coherent to each other (due to Theorem 4), the observable CNF $\boldsymbol{C}^{\blacklozenge}$ is a model of the block $\boldsymbol{B}$ and $\boldsymbol{C}^{\blacklozenge}$ observably covers $\boldsymbol{M}$. ⊗

Lemma 7. Let $\boldsymbol{C}^{\blacklozenge}$ be the observable CNF for a model $\boldsymbol{M}$.

1. If a partial assignment a is conflicting for $\boldsymbol{M}$, then $\boldsymbol{C}^{\blacklozenge}$ contains a clause $\boldsymbol{c}^{\blacklozenge}$, such that $\boldsymbol{c}^{\blacklozenge}(a) = 0$.
2. If a partial assignment a implies an assignment b for $\boldsymbol{M}$, then for each elementary assignment $b_i \in b$ there is a clause in $\boldsymbol{C}^{\blacklozenge}$ that represents an assignment g or an elementary implication $g \Rightarrow b_i$ where $g \subseteq a$ and $b_i \notin a$.
3. If $\boldsymbol{c}^{\blacklozenge} \in \boldsymbol{C}^{\blacklozenge}$, then there exists a partial assignment a, such that there are two possibilities:

a) a is conflicting for $\boldsymbol{M}$ and $c^{\blacklozenge}$ represents a, b) a is implying an elementary assignment b_i for $\boldsymbol{M}$ and $\boldsymbol{c}^{\blacklozenge}$ represents the elementary implication $a \Rightarrow b_i$.

Proof:
1. Let a be conflicting for $\boldsymbol{M}$. Let a clause $\boldsymbol{c}$ represents a in $\boldsymbol{C}^*$. By definition of $\boldsymbol{C}^{\blacklozenge}$ there is $\boldsymbol{c}^{\blacklozenge} \in \boldsymbol{C}^{\blacklozenge}$ such that $\boldsymbol{c}$ covers $\boldsymbol{c}^{\blacklozenge}$. Thus, $\boldsymbol{c}^{\blacklozenge}(a) = 0$.
2. Let a imply an elementary assignment b_i for $\boldsymbol{M}$, and let $\boldsymbol{c}$ be the clause representing the implication $a \Rightarrow b_i$, then by definition of $\boldsymbol{C}^{\blacklozenge}$ there is $\boldsymbol{c}^{\blacklozenge} \in \boldsymbol{C}^{\blacklozenge}$ such that $\boldsymbol{c}$ covers $\boldsymbol{c}^{\blacklozenge}$. There are two cases:
   a) $\boldsymbol{c}^{\blacklozenge}$ contains the variable assigned by b_i. Then $\boldsymbol{c}^{\blacklozenge}$ represents the implication $g \Rightarrow b_i$ where $g \subseteq a$ and $b_i \notin a$.
   b) $\boldsymbol{c}^{\blacklozenge}$ does not contain the variable assigned by b_i. Then $\boldsymbol{c}^{\blacklozenge}(g) = 0$ where $g \subseteq a$ and $b_i \notin a$.
3. Follows by construction of $\boldsymbol{C}^{\blacklozenge}$. ⊗

Lemma 8. Given a model $\boldsymbol{M}$ and its observable and characteristic CNFs $\boldsymbol{C}^{\blacklozenge}$ and $\boldsymbol{C}^{\cdot}$, respectively. For any clause $\boldsymbol{c}^{\blacklozenge} \in \boldsymbol{C}^{\blacklozenge}$ there is a clause $\boldsymbol{c}^{\cdot} \in \boldsymbol{C}^{\cdot}$, such that $\boldsymbol{c}^{\blacklozenge}$ covers $\boldsymbol{c}^{\cdot}$.

Proof: According to Lemma 7, for any clause $\boldsymbol{c}^{\blacklozenge} \in \boldsymbol{C}^{\blacklozenge}$ there exists a partial assignment a, such that a is conflicting for $\boldsymbol{M}$ and $\boldsymbol{c}^{\blacklozenge}$ represents a, or a is implying an elementary assignment b_i for $\boldsymbol{M}$ and $\boldsymbol{c}^{\blacklozenge}$ represents the elementary implication $a \Rightarrow b_i$. In both cases, $\boldsymbol{c}^{\blacklozenge}$ is an implicate of the permission function of the model $\boldsymbol{M}$ by Axiom 5. ⊗

Lemma 9. Let $\boldsymbol{M}$ be a model with maximal implicativity and $\boldsymbol{C}^{\cdot}$ be the characteristic CNF of the model $\boldsymbol{M}$. If a is an implying assignment for $\boldsymbol{C}^{\cdot}$ (under *CNF-BCP*), then a is an implying assignment for $\boldsymbol{M}$. If a is a conflicting assignment for $\boldsymbol{C}^{\cdot}$ (under *CNF-BCP*), then a is a conflicting or an implying assignment for $\boldsymbol{M}$.

Proof: Let a be a conflicting assignment for $\boldsymbol{M}$. According to the second paragraph of the proof of Theorem 3, a is a conflicting assignment for $\boldsymbol{C}^{\cdot}$ (under *CNF-BCP*). Let a be an implying assignment for $\boldsymbol{M}$. The third paragraph of the proof of Theorem 3 delivers that a is a conflicting or an implying assignment for $\boldsymbol{C}^{\cdot}$ (under *CNF-BCP*). Due to Theorem 3, $\boldsymbol{C}^{\cdot}$ (under *CNF-BCP*) has the same (maximal) implicativity as the model $\boldsymbol{M}$. So, for any conflicting or implying assignment a for $\boldsymbol{C}^{\cdot}$ (under *CNF-BCP*) $\boldsymbol{M}$ already provides the classification of a, as Lemma 9 states. ⊗

Lemma 10. Let $\boldsymbol{M}$ be a model with maximal implicativity. Let $\boldsymbol{c}^{\cdot} \in \boldsymbol{C}^{\cdot}$ where $\boldsymbol{C}^{\cdot}$ is the characteristic CNF of the model $\boldsymbol{M}$. Then the assignment a represented by the clause $\boldsymbol{c}^{\cdot}$ is conflicting for the model $\boldsymbol{M}$.

Proof: Denote a as a^0 (we use indexing because later on we will consider a sequence of assignments starting with a). We have $\boldsymbol{c}^{\cdot}(a^0) = 0$. Suppose that a^0 is not conflicting for $\boldsymbol{M}$.

Since $\boldsymbol{c}^{\cdot}(a^0) = 0$, a^0 is conflicting for $\boldsymbol{C}^{\cdot}$ (under *CNF-BCP*), by Lemma 9, a^0 must be implying for $\boldsymbol{M}$ (because it is not conflicting for $\boldsymbol{M}$). Let $\boldsymbol{M}$ produce an implication $a^0 \Rightarrow b$ and $b_0 \in b$ (we use the bottom index for b_0 because it is an elementary assignment, and the top index for a^0 as it can be non elementary). Let $b_0 = \{x_0 = 1\}$ (the case $b_0 = \{x_0 = 0\}$ is considered similarly).

R: Consider the observable CNF $\boldsymbol{C}^{\blacklozenge}$ of the model $\boldsymbol{M}$. We show that there exists a clause $\boldsymbol{c}^{\blacklozenge} \in \boldsymbol{C}^{\blacklozenge}$, such that $\boldsymbol{c}^{\blacklozenge}(a^0) = 0$ or $\boldsymbol{c}^{\blacklozenge}(a^0) = x_0$: Due to Lemma 7, there is a clause $\boldsymbol{c}^{\blacklozenge} \in \boldsymbol{C}^{\blacklozenge}$ such that $\boldsymbol{c}^{\blacklozenge}(g) = 0$ or $\boldsymbol{c}^{\blacklozenge}$ represents an elementary implication $g \Rightarrow b_0$ where $g \subseteq a^0$ and $b_0 \notin a^0$. If $\boldsymbol{c}^{\blacklozenge}(g) = 0$, then $\boldsymbol{c}^{\blacklozenge}(a^0) = 0$. If $\boldsymbol{c}^{\blacklozenge}$ represents $g \Rightarrow b_0$, the clause $\boldsymbol{c}^{\blacklozenge}$ represents the assignment $(g, \neg b_0)$. Then $\boldsymbol{c}^{\blacklozenge}(g) = x_0$. Since $g \subseteq a^0$ and $b_0 \notin a^0$, we have $\boldsymbol{c}^{\blacklozenge}(a^0) = x_0$.

Consider the assignment (a^0, b_0). Since $\boldsymbol{c}^{\cdot}(a^0) = 0$, (a^0, b_0) is conflicting for $\boldsymbol{C}^{\cdot}$ (under *CNF-BCP*). Due to Lemma 9 (a^0, b_0) must be conflicting or implying for $\boldsymbol{M}$.

First, we give evidence that (a^0, b_0) cannot be conflicting for $\boldsymbol{M}$. Suppose, the inverse statement is true, that (a^0, b_0) is conflicting for $\boldsymbol{M}$. By Lemma 7, $\boldsymbol{C}^{\blacklozenge}$ must contain a clause $\boldsymbol{c}$, such that $\boldsymbol{c}(a^0, b_0) = 0$. Recall that that b_0 assigns a value to a variable x_0. There are two cases: 1) $\boldsymbol{c}$ contains the variable x_0, 2) $\boldsymbol{c}$ does not contain the variable x_0.
1. If $\boldsymbol{c}^{\blacklozenge}(a^0) = 0$, then a^0 is conflicting for $\boldsymbol{C}^{\blacklozenge}$ under *CNF-BCP*.

Let $\boldsymbol{c}^{\blacklozenge}(a^0) = x_0$, then $\boldsymbol{C}^{\blacklozenge}$ contains two clauses $\boldsymbol{c}^{\blacklozenge}$ and $\boldsymbol{c}$, which under assignment a^0, are turned into complementary unit clauses x_0 and $\neg x_0$. Hence, the assignment a^0 must be classified as conflicting under *CNF-BCP*(a^0) for the observable CNF $\boldsymbol{C}^{\blacklozenge}$.

However, a^0 was proven to be implying for $\boldsymbol{C}^{\blacklozenge}$.

2. We have $\boldsymbol{c}(a^0) = 0$, hence again a^0 must be classified as conflicting under *CNF-BCP*(a) for $\boldsymbol{C}^{\blacklozenge}$, which is impossible.

So, the assignment (a^0, b_0) must be implying for $\boldsymbol{M}$.

Let the assignment (a^0, b_0) imply an elementary assignment b_1 for $\boldsymbol{M}$. Now we would like to prove that (a^0, b_0, b_1) is implying for $\boldsymbol{M}$. Take into account that $\boldsymbol{c}^{\cdot}(a^0, b_0) = 0$, since $\boldsymbol{c}^{\cdot}(a^0) = 0$. Now we can repeat the previous proof for (a^0, b_0) (starting at the paragraph labeled with R) by replacing a^0 with $a^1 = (a^0, b_0)$, and b_0 with b_1, and the variable x_0 assigned by b_0 with a variable x_1 assigned by b_1.

Going on in such a way we will construct an infinite sequence of implying assignments of the sort (a^0, b_0 , b_1 , …, b_k). This is impossible, since the number of variables is finite by Axiom 1. ⊗

Lemma 11. Let $\boldsymbol{M}$ be a model with maximal implicativity. Then the observable CNF $\boldsymbol{C}^{\blacklozenge}$ and the characteristic CNF $\boldsymbol{C}^{\cdot}$ of the model are identical.

Proof:

1. Suppose, there exists a clause $\boldsymbol{c}^{\blacklozenge} \in \boldsymbol{C}^{\blacklozenge} \setminus \boldsymbol{C}^{\cdot}$. Due to Lemma 8, there exists a clause $\boldsymbol{c}^{\cdot} \in \boldsymbol{C}^{\cdot}$, such that $\boldsymbol{c}^{\blacklozenge}$ covers $\boldsymbol{c}^{\cdot}$. Thus, there must be a literal in $\boldsymbol{c}^{\blacklozenge}$ (say a literal x) that is not contained in $\boldsymbol{c}^{\cdot}$. Let (g, b_0) be the assignment represented by clause $\boldsymbol{c}^{\blacklozenge}$, where $b_0 = \{x = 0\}$. By this construction $\boldsymbol{c}^{\cdot}(g) = 0$.
   Let's give evidence first that the assignment g cannot be conflicting for $\boldsymbol{M}$. Otherwise, Lemma 7 delivers a clause $\boldsymbol{c} \in \boldsymbol{C}^{\blacklozenge}$, such that $\boldsymbol{c}(g) = 0$. Hence, there exists a clause in $\boldsymbol{C}^{\blacklozenge}$ (namely, the clause $\boldsymbol{c}$) that is covered by $\boldsymbol{c}^{\blacklozenge}$(which is not equal to $\boldsymbol{c}^{\blacklozenge}$). This is impossible for an observable CNF by its definition. Thus, g is not conflicting for $\boldsymbol{M}$.
   Now consider the assignment a represented by $\boldsymbol{c}^{\cdot}$. By Lemma 10, a is conflicting for the model $\boldsymbol{M}$. Since $\boldsymbol{c}^{\cdot}(g) = 0$, $a \subseteq g$. As long as $a \subseteq g$, the assignment g is also conflicting for $\boldsymbol{M}$, which is impossible. Thus, $\boldsymbol{C}^{\blacklozenge} \subseteq \boldsymbol{C}^{\cdot}$.
2. Suppose there exists a clause $\boldsymbol{c}^{\cdot} \in \boldsymbol{C}^{\cdot} \setminus \boldsymbol{C}^{\blacklozenge}$. Due to Lemma 10, the assignment a represented by the clause $c^{\cdot}$ is conflicting for $\boldsymbol{M}$. Thus by Lemma 7, $\boldsymbol{C}^{\blacklozenge}$ contains a clause $\boldsymbol{c}$, such that $\boldsymbol{c}(a) = 0$. Then $c^{\cdot}$ covers $\boldsymbol{c}$. The only possibility is $\boldsymbol{c}^{\cdot} = \boldsymbol{c}$, since both $\boldsymbol{c}$ and $\boldsymbol{c}^{\cdot}$ are implicates of the permission function of the model $\boldsymbol{M}$, but $\boldsymbol{c}^{\cdot}$ is a prime one. Hence $\boldsymbol{c}^{\cdot}$ is contained in $\boldsymbol{C}^{\blacklozenge}$ that contradicts to the supposition. Thus, $\boldsymbol{C}^{\blacklozenge}$ and $\boldsymbol{C}^{\cdot}$ are identical. ⊗

Proof of Theorem 6:

⇒: Consider the characteristic CNF $\boldsymbol{C}^{\cdot}$ of the block $\boldsymbol{B}$. Let $\boldsymbol{y}$ implement the constant function d. Then a unit clause $\boldsymbol{c}$ that represents assignment $\neg a = \{ \boldsymbol{y} = \neg d \}$ is an implicate of the permission function $\boldsymbol{f}$ of the block $\boldsymbol{B}$. According to Theorem 1, any implicate of the permission function $\boldsymbol{f}$ must contain at least one output variable of the block $\boldsymbol{B}$. Since the clause $\boldsymbol{c}$ contains only one variable, it is a prime implicate of $\boldsymbol{f}$. Hence $\boldsymbol{c}$ is contained in the characteristic CNF $\boldsymbol{C}^{\cdot}$. Since $\boldsymbol{C}^{\cdot}$ is identical to the observable CNF $\boldsymbol{C}^{\blacklozenge}$ of the model $\boldsymbol{M}$ (due to Lemma 10), $\boldsymbol{c} \in \boldsymbol{C}^{\blacklozenge}$.
There are only two possible reasons why the unit clause $\boldsymbol{c}$ is contained in the observable CNF $\boldsymbol{C}^{\blacklozenge}$: 1) The assignment $\neg a = \{ \boldsymbol{y} = \neg d \}$ is conflicting for the model $\boldsymbol{M}$ or 2) the empty assignment g implies a.

⇐: Let $\boldsymbol{y}$ not implement the constant function d. Suppose the theorem does not hold, i.e. the elementary assignment $\neg a = \{ \boldsymbol{y} = \neg d \}$ is conflicting or the empty assignment g implies $a = \{ \boldsymbol{y} = d \}$ for the model $\boldsymbol{M}$. In both cases the unit clause $\boldsymbol{c}$ representing assignment $\neg a = \{ \boldsymbol{y} = \neg d \}$ is an implicate of the permission function $\boldsymbol{f}$. According to Theorem 1, any implicate of the permission function $\boldsymbol{f}$ must contain at least one output variable of the block $\boldsymbol{B}$. Thus, the unit clause $\boldsymbol{c}$ is a prime implicate of $\boldsymbol{f}$. At the same time by Lemma 1, for any full assignment b to the input variables of the block $\boldsymbol{B}$ there is exactly one full assignment to the output variables of the block, such that $\boldsymbol{f}(a, b) = 1$. Hence, the block $\boldsymbol{B}$ must implement the constant Boolean function d on the output y. This contradicts to the supposition. ⊗

Proof of Theorem 7: Consider an elementary implication $a \Rightarrow b_i$ for $\boldsymbol{M}$. According to Axiom 5 the clause $\boldsymbol{c}$ representing the assignment $a \cup \neg b_i$ is an implicate of the permission function of the block $\boldsymbol{B}$. Thus, $\boldsymbol{c}$ covers a prime implicate $\boldsymbol{c}^{\cdot}$ contained in $\boldsymbol{C}^{\cdot}$. The assignment $a \cup \neg b_i$ falsifies $\boldsymbol{c}^{\cdot}$ and hence it is conflicting for $\boldsymbol{M}$.

Let an assignment a be conflicting for $\boldsymbol{M}$, and let $a' = a \setminus b_i$ where b_i is an elementary assignment from a. If a' is conflicting for $\boldsymbol{M}$, then condition 2 of consistency holds. Let a' be not conflicting for $\boldsymbol{M}$. Then there is the prime implicate $\boldsymbol{c}^{\cdot} \in \boldsymbol{C}^{\cdot}$ representing the assignment $a = a' \cup b$. Thus, $a' \Rightarrow \neg b_i$ for $\boldsymbol{M}$. ⊗

Lemma 12. Let $\boldsymbol{M}$ be a consistent model of a block $\boldsymbol{B}$, and let a' and a'' be some conflicting assignments for $\boldsymbol{M}$ such that the clauses $\boldsymbol{c}'$ and $\boldsymbol{c}''$ representing a' and a'' accordingly can be resolved. Then the resolvent $\boldsymbol{c}$ represents a conflicting assignment for $\boldsymbol{M}$.

Proof: Let $a' = g' \cup b_i$ and $a'' = g'' \cup \neg b_i$ where b_i is an elementary assignment assigning a value to a variable by which $\boldsymbol{c}'$ and $\boldsymbol{c}''$ are resolved. Then their resolvent $\boldsymbol{c}$ represents the assignment $g' \cup g''$. As a' and a'' are conflicting and $\boldsymbol{M}$ is consistent, $g' \Rightarrow \neg b_i$ and $g'' \Rightarrow b_i$. Due to Axiom 4 (on monotony), as $g' \Rightarrow \neg b_i$, $g' \cup g''$ is conflicting or $g' \cup g'' \Rightarrow \neg b_i$. Analogically, as $g'' \Rightarrow b_i$, $g' \cup g''$ is conflicting or $g' \cup g'' \Rightarrow b_i$. If $g' \cup g''$ is not conflicting, then we have $g' \cup g'' \Rightarrow \neg b_i$ and $g' \cup g'' \Rightarrow b_i$ for $\boldsymbol{M}$ simultaneously that contradicts to Axiom 3. Thus, $g' \cup g''$ is conflicting. ⊗

Proof of Theorem 8: Let g be a complete assignment to the pin variables of $\boldsymbol{B}$ which falsifies the permission function $\boldsymbol{f}$ of $\boldsymbol{B}$. Let a be a full assignment to the inputs of $\boldsymbol{B}$ where $a \subseteq g$. Due to Axiom 6 the assignment a implies a full assignment b to the outputs of $\boldsymbol{B}$. By lemma 1 the assignment g must contain at least one elementary assignment $\neg b_i$ such that b_i is contained in b. Hence $a \cup \neg b_i \subseteq g$. As $a \Rightarrow b_i$ for $\boldsymbol{M}$ and $\boldsymbol{M}$ is consistent, $a \cup \neg b_i$ is conflicting for $\boldsymbol{M}$. Due to Axiom 4 (on monotony) g is conflicting for $\boldsymbol{M}$. ⊗

Lemma 13. Let $\boldsymbol{M}$ be a consistent and recognizing maximal conflicts model of a block $\boldsymbol{B}$. Let a clause $\boldsymbol{c}$ be an implicate of the permission function $\boldsymbol{f}$ of $\boldsymbol{B}$. Then the assignment a represented by $\boldsymbol{c}$ is conflicting for $\boldsymbol{M}$.

Proof: As the clause $\boldsymbol{c}$ is an implicate of $\boldsymbol{f}$, it can be constructed by resolving some clauses representing complete assignments falsifying $\boldsymbol{f}$. At the same time all these assignments are conflicting, as $\boldsymbol{M}$ is recognizing maximal conflicts. Hence, by Lemma 12 the assignment a represented by $\boldsymbol{c}$ is conflicting for $\boldsymbol{M}$. ⊗

Proof of Theorem 9: Let an assignment a be classified as conflicting by some model of $\boldsymbol{B}$. Then the clause $\boldsymbol{c}$ representing a is an implicate of the permission function $\boldsymbol{f}$ of $\boldsymbol{B}$ (by Axiom 5). Then the assignment a is conflicting for $\boldsymbol{M}$ due to Lemma 13.

Let there exist an elementary implication $a \Rightarrow b_i$ in some model of $\boldsymbol{B}$. Then the clause $\boldsymbol{c}$ representing the assignment $a \cup \neg b_i$ is an implicate of the permission function $\boldsymbol{f}$ of $\boldsymbol{B}$ (by Axiom 5). Thus, $a \cup \neg b_i$ is conflicting for $\boldsymbol{M}$ due to Lemma 13. As the model $\boldsymbol{M}$ is consistent, $a \Rightarrow b_i$ or a is conflicting for $\boldsymbol{M}$. Thus, $\boldsymbol{M}$ does not have a lower implicativity than any other model of $\boldsymbol{B}$. ⊗

Proof of Theorem 10: Follows from Theorem 8 and Theorem 9. ⊗

Lemma 14. Let $\boldsymbol{M}$ be a model with maximal implicativity of a block $\boldsymbol{B}$, and let $\boldsymbol{C}^{\blacklozenge}$ be the observable CNF of the model $\boldsymbol{M}$. Both $\boldsymbol{M}$ and $\boldsymbol{C}^{\blacklozenge}$ (under *CNF-BCP*) provide the same classification of any assignment a to the pin variables of $\boldsymbol{B}$.

Proof: By lemma 11 $\boldsymbol{C}^{\blacklozenge}$ is identical to the characteristic CNF $\boldsymbol{C}^{\cdot}$ of $\boldsymbol{M}$. Due to lemma 9, an implying assignment a for $\boldsymbol{C}^{\blacklozenge}$ is implying for $\boldsymbol{M}$. Let us consider a conflicting assignment a for $\boldsymbol{C}^{\blacklozenge}$. By lemma 9 a can be conflicting or implying for $\boldsymbol{M}$.

Suppose a is implying for $\boldsymbol{M}$. As a is conflicting for $\boldsymbol{C}^{\cdot}$, there is a prime implicate $\boldsymbol{c}^{\cdot} \in \boldsymbol{C}^{\cdot}$ such that $\boldsymbol{c}^{\cdot}(a) = 0$. Let $a' \subseteq a$ be the assignment represented by the clause $\boldsymbol{c}^{\cdot}$. As $\boldsymbol{c}^{\cdot}(a') = 0$, a' is conflicting for $\boldsymbol{C}^{\cdot}$ under *CNF-BCP*. By Lemma 10 a' is conflicting for $\boldsymbol{M}$. As $a' \subseteq a$, a must be conflicting for $\boldsymbol{M}$ by Axiom 4 (on monotony), that contradicts to the supposition. ⊗

Lemma 15. Let a consistent model $\boldsymbol{M}$ have maximal implicativity. Then $\boldsymbol{M}$ is observably coherent to its observable CNF $\boldsymbol{C}^{\blacklozenge}$.

Proof: By Lemma 14 $\boldsymbol{M}$ and $\boldsymbol{C}^{\blacklozenge}$ provide identical classification of all assignments. We need to prove that $g = g'$, if $a \Rightarrow g$ for $\boldsymbol{M}$ and $a \Rightarrow g'$ for $\boldsymbol{C}^{\blacklozenge}$. Suppose $g \neq g'$. There are two cases:

1. There is an elementary assignment b_i such that $b_i \in g \setminus g'$.
2. There is an elementary assignment b_i such that $b_i \in g' \setminus g$.

1. As $\alpha \Rightarrow b_i$ for $\boldsymbol{M}$, $\alpha \cup \neg b_i$ is conflicting for $\boldsymbol{M}$ due to consistency of $\boldsymbol{M}$. Then $\alpha \cup \neg b_i$ is conflicting for $\boldsymbol{C}^\blacklozenge$ by Lemma 14. By lemma 11 $\boldsymbol{C}^\blacklozenge$ is identical to the characteristic CNF $\boldsymbol{C}^{\cdot}$ of $\boldsymbol{M}$. Thus, $\boldsymbol{C}^\blacklozenge$ is a consistent model due to Theorem 7. Hence, as $\alpha \cup \neg b_i$ is conflicting and α is implying for $\boldsymbol{C}^\blacklozenge$, $\alpha \Rightarrow b_i$ for $\boldsymbol{C}^\blacklozenge$. Hence $b_i \in \gamma$ and $b_i \notin \gamma \setminus \gamma$.
2. Analogically to case 1. ⊗

Lemma 16. If models $\boldsymbol{M_1}$ and $\boldsymbol{M_2}$ of a block $\boldsymbol{B}$ have maximal implicativity, their observable CNFs $\boldsymbol{C_1}^\blacklozenge$ and $\boldsymbol{C_2}^\blacklozenge$ are observably coherent (under CNF-BCP).

Proof: By Lemma 11, $\boldsymbol{C_1}^\blacklozenge$ and $\boldsymbol{C_2}^\blacklozenge$ are identical. ⊗

Proof of Theorem 11: Follows from Lemmas 15 and 16. ⊗

Proof of Theorem 12: Follows from Theorems 10 and 11. ⊗

Proof of Theorem 13: By Lemmas 14 and 16 any two models with maximal implicativity provide the same classification of partial assignments. Suppose the model $\boldsymbol{M}$ has not maximal implicativity. Thus, there is an assignment α which is neither conflicting nor implying for $\boldsymbol{M}$ but is conflicting or implying for any model $\boldsymbol{M'}$ with maximal implicativity for the block $\boldsymbol{B}$.

Let α be conflicting for $\boldsymbol{M'}$ (but not conflicting or implying for $\boldsymbol{M}$). Consider the set of all partial assignments covering α and select and assignment α' from this set such that α' is not conflicting for $\boldsymbol{M}$ while any assignment covering α' is conflicting for $\boldsymbol{M}$. Such an assignment α' must exist as the number of the pin variables of the block $\boldsymbol{B}$ is finite (Axiom 1). Now we can extend the model $\boldsymbol{M}$ by providing possibility to classify α' as conflicting (this is consistent with Axiom 4 on monotony). However this is impossible as the model $\boldsymbol{M}$ has maximal strong implicativity.

The case when α is implying for $\boldsymbol{M'}$ is considered analogically. ⊗

Proof of Theorem 14: Let (α, b_i) be conflicting for $\boldsymbol{M}$ where b_i is an elementary assignment. Suppose that α is not conflicting for $\boldsymbol{M}$. First we show that α must imply $\neg b_i$ for $\boldsymbol{M}$. This is indeed the case, otherwise, due to the finite number of the pin variables of the block $\boldsymbol{B}$, there would be an assignment α' covering α ($\alpha \subseteq \alpha'$) such that any assignment γ covering α' and containing neither b_i nor $\neg b_i$, which is conflicting or implying $\neg b_i$. In this case, we could extend the model $\boldsymbol{M}$ by providing the ability to produce the implication $\gamma \Rightarrow \neg b_i$. But this contradicts to the maximum of strong implicativity for $\boldsymbol{M}$. Thus, α is conflicting or $\alpha \Rightarrow \neg b_i$.

Let $\alpha \Rightarrow b_i$ for $\boldsymbol{M}$. If $(\alpha, \neg b_i)$ is not conflicting for $\boldsymbol{M}$, then, due to the finite number of the pin variables of the block $\boldsymbol{B}$, there is an assignment α' covering $(\alpha, \neg b_i)$ such that α' is not conflicting for $\boldsymbol{M'}$ while any assignment covering α' is conflicting for $\boldsymbol{M'}$. In this case, we can extend the model $\boldsymbol{M}$ by providing possibility to classify α' as conflicting. However, this is impossible as the model $\boldsymbol{M}$ has maximal strong implicativity. Thus, $(\alpha, \neg b_i)$ is conflicting for $\boldsymbol{M}$.

Thus, the model $\boldsymbol{M}$ is consistent. ⊗

Proof of Theorem 15: Follows from Theorems 11, 13, and 14. ⊗

Proof of Theorem 16: Let $\boldsymbol{M'}$ be a consistent model with maximal strong implicativity for the block $\boldsymbol{B}$. By Theorem 13, $\boldsymbol{M'}$ has maximal implicativity. By Theorem 11 $\boldsymbol{M}$ and $\boldsymbol{M'}$ are observably coherent. ⊗

Proof of Theorem 17: Follows from Theorem 14, Theorem 10 and Theorem 16. ⊗

Proof of Theorem 18: The procedure differs from the procedure of the exhaustive simulation used for definition of the observable CNF in Section 5 only in that some partial assignments are not affecting the block B (as a consequence of conflicts and backtracks). However for each such assignment α there is an assignment $\alpha' \subset \alpha$ that is classified as conflicting during the procedure *CONSTRUCT_OBS*. Hence α is also conflicting, and the clause $\boldsymbol{c}$ representing it is to be added to the observable CNF that is constructed by the procedure of exhaustive simulation. However after that, due to the definition of the observable CNF in Section 5 the clause $\boldsymbol{c}$ will be removed from the CNF, since it covers the clause $\boldsymbol{c'}$ representing the assignment α'. Thus, both procedures construct identical CNFs. ⊗

Proof of Theorem 19: Follows from the method of Blake-Poretski [29,30]. ⊗

Proof of Theorem 20: By construction, the resulting CNF $\boldsymbol{C}^{\cdot}$ represents the permission function $\boldsymbol{f}$ of the block $\boldsymbol{B}$.

Let c' and c'' be two arbitrary clauses of the resulting CNF $C^{\cdot}$, such that c' and c'' can be resolved producing the resolvent c. Due to Theorem 19, it suffices to prove that c covers a clause from $C^{\cdot}$.

Let c' and c'' be resolved by a variable u, and let the resolvent c represent an assignment a. Let c' contain u and c'' contain $\neg u$. Consider the assignment $a \cup \{ u = 0 \}$ (that unsatisfies c' and c). Since the branches "-" are always examined by the procedure after the branches "0" and "1", the procedure must pass along a path on which all definite decision assignments belong to the assignment $a \cup \{ u = 0 \}$. We say the path to be corresponding to the assignment $a \cup \{ u = 0 \}$.

Now, we show that, on this path, the procedure must deduce a clause c^* representing an assignment $a^* \subseteq a \cup \{ u = 0 \}$. If the procedure encounters a conflict on the path, then it must add such clause c^* to the current CNF C^*. Let the procedure has passed through the path and continues the search in depth by constructing the child node for the last node N of the path. Take into account that c^* is an implicate of the permission function f in the case that $a^* = a \cup \{ u = 0 \}$, because c^* covers c' and c' is an implicate of f. Hence, due to conflict inheritance, the procedure must mark the node N to be conflicting and must add the clause c^* (for $a^* = a \cup \{ u = 0 \}$) to the current CNF C^*.

In the same way, one can prove that on a path corresponding to the assignment $a \cup \{ u = 1 \}$ the procedure must deduce a clause c^{**} representing an assignment $a^{**} \subseteq a \cup \{ u = 1 \}$.

If the clause c^* or c^{**} does not contain the variable u, the clause is covered by the resolvent c. Thus, the resulting CNF $C^{\cdot}$ contains the clause covered by the resolvent c. Let both c^* or c^{**} contain the variable u (in this case, c^* contains the literal u and c^{**} contains the literal $\neg u$). Since the branches "-" are examined in the last turn, the procedure will pass along a path corresponding to the assignment a after c^* and c^{**} have been already added to the current CNF C^*. Hence, the procedure will inevitably encounter a conflict on this path (due to clauses c^* and c^{**}) and add a clause that is covered by the resolvent c to the current CNF C^*. ⊗

Proof of Theorem 21: First, we give evidence that $f(x,y)$ implies $\exists z\, f^*(x,y,z)$:

Let a be a complete assignment to the input variables x, and let $y = Y(x)$ be the function implemented by the system S. Consider the complete assignment b to the output variables y of S that is the reaction of S under a, i.e. $b = Y(a)$. The assignment (a, b) satisfies the permission function $f(x,y)$.

Under the assignment a to the inputs, the internal variables z of S take the assignment g which can be calculated in accordance to functions Y_i implemented by the normal blocks B_i of the system. (The calculation of g follows the topological order of the blocks B_i, where block-constraints being activated cannot constrain the signal propagation due to the fitting axiom). Thus, the assignment (a, b, g) must satisfy the extended permission function $f^*(x,y,z)$. Hence, the assignment (a, b) satisfies the cofactor $f^*_{z=g}$. Due to formula (1), the assignment (a, b) satisfies $\exists z\, f^*(x,y,z)$. Thus, for any assignment (a, b) satisfying $f(x,y)$ we have proven that the same assignment satisfies $\exists z\, f^*(x,y,z)$, i.e. $f(x,y)$ implies $\exists z\, f^*(x,y,z)$.

Now, we show that any assignment (a, b') unsatisfying $f(x,y)$ does not satisfy $\exists z\, f^*(x,y,z)$: If (a, b') does not satisfy $f(x,y)$, then $b' \neq b$ where b is the correct reaction of the system under a, i.e. $b = Y(a)$. Then for any assignment g to the internal variables z of the system S, the permission function f^i of at least one normal block B^i must be unsatisfied, and the extended permission function $f^*(x,y,z)$ takes the value 0 under the assignment (a, b', g). Hence the cofactor $f^*_{z=g}$ takes the same value under the assignment (a, b'). Since all cofactors $f^*_{z=g}$ take value 0 under (a, b'), formula (1) delivers that $\exists z\, f^*(x,y,z)$ is equal to 0 under (a, b'). ⊗

Proof of Theorem 22: Consider all elements of the list *DEDUCED* which were marked during the procedure *REVERSE-BCP*(g) and whose direct reasons are not the empty assignment. A sequence of these elements, ordered in the direction from the end of the list *DEDUCED* to its beginning, is called *marked sequence*. (In Example 9, the marked sequence is $f = 0$, $f = 1$, $d = 1$)

Let $Q = c_1, c_2, \ldots, c_p$ be a marked sequence. Let e_i be a direct reason for c_i (i = 1,…,p) that is saved in the list *REASONS*. Consider an elementary assignment $g_j \in g$. Let $Q(g_j)$ be the subsequence $c_{j_1}, c_{j_2}, \ldots, c_{j_p}$ of Q that starts with $c_{j_1} = g_j$ and which consists of all elements c_{j_k} ($j_1 < j_k$) of Q such that c_{j_k} is contained in the reason e_{j_l} of some previous element c_{j_l} ($j_l < j_k$) of the subsequence. Conceptually, the subsequence $Q(g_j)$ describes an implication chain for deriving g_j, and it is called *reasoning subsequence* for g_j. (In Example 9, the reasoning subsequence $Q(f = 0)$ is $f = 0$, $d = 1$).

For each element c_{j_k} of the reasoning subsequence $Q(g_j)$ consider the clause $c(c_{j_k})$ representing the elementary implication $e_{j_k} \Rightarrow c_{j_k}$ where e_{j_k} is the direct reason for c_{j_k} saved in the list *REASONS*. The

clause $\boldsymbol{c}(c_{j_k})$ is an implicate of the permission function $\boldsymbol{f}_{j_k}$ of the block $\boldsymbol{B}_{j_k}$ which was used to produce the implication $e_{j_k} \Rightarrow c_{j_k}$. Consequently, $\boldsymbol{c}(c_{j_k})$ is an implicate of the extended permission function $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},g)$.

Given a subsequence $\boldsymbol{Q}(g_j) = c_{j_1}, c_{j_2},\ldots, c_{j_p}$ consider the sequence of clauses $\boldsymbol{c}(c_{j_1})$, $\boldsymbol{c}(c_{j_2})$,…, $\boldsymbol{c}(c_{j_p})$. (In Example 9, for $\boldsymbol{Q}(f = 0)$ the sequence is $\neg d \vee y^1 \vee \neg f,\quad \neg b \vee d.$) By construction the sequence is "resolvable" in a sense that that the first clause $\boldsymbol{c}(c_{j_1})$ can be resolved with the second $\boldsymbol{c}(c_{j_2})$, after that the product of resolution can be resolved with the next clause $\boldsymbol{c}(c_{j_3})$, and so on. Let $\boldsymbol{c}(Q(g_j))$ be a product of resolution of the considered sequence of clauses. Since all clauses $\boldsymbol{c}(c_{j_k})$ of the sequence are implicates of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},g)$, the product $\boldsymbol{c}(Q(g_j))$ is also implicate of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},g)$. (In Example 9, $\boldsymbol{c}(\boldsymbol{Q}(f = 0)) = \neg b \vee y^1 \vee \neg f$).

On the other hand, according to construction the resolvent $\boldsymbol{c}(Q(g_j))$ represents an elementary implication $b \Rightarrow g_j$ where the partial assignment b consists of elementary assignments which belong to direct reasons of an assignment from $Q(g_j)$ and at the same time are contained in a.

If g is an elementary assignment, then the marked sequence $\boldsymbol{Q}$ is the same as its reasoning subsequence $Q(g)$, and the clause representing $b \Rightarrow g$ is an implicate of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},g)$ because it is exactly the resolvent $\boldsymbol{c}(Q(g))$. Thus, the case 1 of the theorem is proven.

Suppose now that the clause $\boldsymbol{c}(g)$ representing the assignment g is an implicate of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},g)$. For each $g_j \in g$ the resolvent $\boldsymbol{c}(Q(g_j))$ is also implicate of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},g)$, in addition the clause $\boldsymbol{c}(Q(g_j))$ represents the implication $b\natural \Rightarrow g_j$ where $b\natural \subseteq a$. Now the clauses $\boldsymbol{c}(g)$ and $\boldsymbol{c}(Q(g_j))$ can de resolved, and by construction of the reasoning subsequences $Q(g_j)$ the resolvent $\boldsymbol{c}$ represents the assignment b delivered by the procedure *REVERSE-BCP*(g). Since the resolved clauses are implicates of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},g)$, the resolvent $\boldsymbol{c}$ is an implicate of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},g)$. Thus, the case 2 of the theorem is proven. ⊗

Lemma 17. Let a Boolean function $j(\boldsymbol{x},\boldsymbol{z})$ imply a Boolean function $j\natural(\boldsymbol{x},z)$, i.e. $(j(\boldsymbol{x},\boldsymbol{z}) \rightarrow j\natural(\boldsymbol{x},z)) = 1$. Then $\exists z\, j(\boldsymbol{x},\boldsymbol{z}) \rightarrow \exists z\, j\natural(\boldsymbol{x},z) = 1$.

Proof: Since $j(\boldsymbol{x},\boldsymbol{z}) \rightarrow j\natural(\boldsymbol{x},z) = 1$, for each assignment $g \in 2^{\boldsymbol{z}}$ the cofactors $j_{\boldsymbol{z}=g}$ and $j\natural_{\boldsymbol{z}=g}$ have to satisfy to the same relation, i.e. $j_{\boldsymbol{z}=g} \rightarrow j\natural_{\boldsymbol{z}=g} = 1$. Hence formula (1) (given in the Section 9.2) delivers $\exists z\, j(\boldsymbol{x},\boldsymbol{z}) \rightarrow \exists z\, j\natural(\boldsymbol{x},z) = 1$. ⊗

Lemma 18. Let the procedure *SBCP*(a) classify the assignment a to be conflicting for a system $\boldsymbol{S}$ having the permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$. Then the clause representing a is an implicate of $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$.

Proof: The direct reason of the conflict can be of two types. First, it is a pair $g = b_i, \neg b_i$ of opposite elementary assignments. Second, it is a conflicting assignment g for a block of the system $\boldsymbol{S}$. In any case, the clause $\boldsymbol{c}^*$ representing the direct reason g of the conflict is an implicate of the extended permission function $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ of the system $\boldsymbol{S}$ where $\boldsymbol{z}$ is a vector of internal variables of the system. (In the first case, the representing clause $\boldsymbol{c}^*$ is equivalent to the constant Boolean function 1. In the second case, the clause $\boldsymbol{c}^*$ is an implicate of the permission function $\boldsymbol{f}_j$ of the block for which the assignment g is conflicting.) Consider the indirect reason $a\natural$ of the conflict that is produced by the procedure *REVERSE-BCP*(g). According to the procedure $a\natural \subseteq a$ and by Theorem 22 the clause $\boldsymbol{c}\natural$ representing $a\natural$ is an implicate of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$. Since $a\natural \subseteq a$, the clause $\boldsymbol{c}$ representing a is also implicate of $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$. By Lemma 17, $\exists z\, \boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) \rightarrow \exists z\, \boldsymbol{c} = 1$ (i.e. after existential quantification on z the clause $\exists z\, \boldsymbol{c}$ is still implicate of $\exists z\, \boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$) . As the clause $\boldsymbol{c}$ does not depend on internal variables z, $\exists z\, \boldsymbol{c} = \boldsymbol{c}$. On the other hand, by Theorem 13 $\exists z\, \boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$. Finally, we have $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y}) \rightarrow \boldsymbol{c} = 1$. ⊗

Lemma 19. Let the procedure *SBCP*(a) classify the assignment a to be implying an assignment b for a system $\boldsymbol{S}$ having the permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$. Then for each elementary assignment $b_j \in b$ the clause representing the elementary implication $a \Rightarrow b_j$ is an implicate of $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$.

Proof: Similar to that of Lemma 18, however, instead of Theorem 22.1 one has to refer to Theorem 22.2. ⊗

Lemma 20. The procedure *SBCP*(a) satisfies Axiom 5 of a model.

Proof: Follows from Lemmas 18 and 19. ⊗

Lemma 21. Let $\boldsymbol{S}$ be a system implementing a Boolean vector function $\boldsymbol{y} = Y(\boldsymbol{x})$. For each value assignment a to all input variables of the system, the procedure $SBCP(a)$ deduces a value assignment b to all output variables of the system, such that $b = Y(a)$.

Proof: Note, that under the full assignment a to input variables of the system the procedure $SBCP(a)$ cannot encounter a conflict. Otherwise, due to Lemma 18, the clause $\boldsymbol{c}$ representing a would have to be an implicate of the permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ of the system. This is impossible, since according to Theorem 1 any implicate of the permission function must contain at least one output variable of the system. (Note, Theorem 1 was proven for a normal block, however the proof was based only on Lemma 1 which is correct for a system, hence Theorem 1 is correct for a system too.)

According to the topology of the system as combinatorial circuit and due to Axiom 6 for system normal blocks, all variables of the system take values as a result of the procedure $SBCP(a)$. Let $\boldsymbol{x} = a$, $\boldsymbol{y} = b$, $\boldsymbol{z} = g$ under the procedure ($\boldsymbol{z}$ is the vector of internal variables of the system). The assignment (a, b, $\boldsymbol{z}$) must satisfy the extended permission function $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ of the model, because, due to Axiom 5 it satisfies the permission function of each normal block of the system and by the fitting axiom it satisfies the permission function of each block-constraint of the system. Thus, the assignment (a, b) satisfies the permission function $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ of the system (because $\exists z\, \boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ by Theorem 13). This is possible iff $b = Y(a)$. ⊗

Lemma 22. The procedure $SBCP(a)$ satisfies Axiom 4 on monotone classification of a model.
Proof: Let a is classified by *SBCP* to be conflicting. Let a' be a partial assignment extending a to the pin variables of the block under consideration. Note, that *SBCP* accumulate assignments to variables in the list *DEDUCED*. Starting *SBCP* with a' instead of a can only lead to adding additional assignments to the list *DEDUCED* at each step of the procedure. Due to Axiom 4 for system blocks this can lead to an earlier conflict or to encountering the same conflict as for $SBCP(a)$. Thus, a' is classified by *SBCP* as conflicting. The monotone classification of implying assignments (as it is needed for Axiom 4) is proven analogically.

Now we show that for any elementary implication $a \Rightarrow b_i$ the assignment $a \cup b_i$ is not conflicting. Let $a \Rightarrow b$ and $b_i \in b$. As $a \Rightarrow b$, then $a \cup b$ is not conflicting, because the procedure $SBCP(a)$ is finished in a situation when the pin variables of the system under consideration are assigned to $a \cup b$ and there is no conflict. Then $a \cup b_i$ can not be conflicting, as otherwise $a \cup b$ must be conflicting due to monotone classification of conflicting assignments proven above. ⊗

Proof of Theorem 23: Follows from Lemma 19 through 22. ⊗

Proof of Theorem 24: By Theorem 23 and $\boldsymbol{S}$ is a normal block.

Let $\boldsymbol{A}$ deliver a counterexample $g = a \cup \{y = \neg d\}$. Let us show that $y(a) = \neg d$. Suppose, that $y(a) = d$. Let y be the vector function implemented by $\boldsymbol{S}$. As a is a full assignment to the inputs of the system $\boldsymbol{S}$, $a \Rightarrow b$ due to Axiom 6 where b is the full assignment to the outputs of $\boldsymbol{S}$ such that $b = y(a)$. Thus, $\{y = d\} \in b$ and $a \Rightarrow \{y = d\}$. Hence $SBCP(a)$ adds the assignment $\{y = d\}$ in the list *DEDUCED*. Then $SBCP(a \cup \{y = \neg d\})$ as well has to add the assignment $\{y = d\}$ in the list *DEDUCED*. Indeed, on the one hand, this procedure can not remove elements from the list *DEDUCED*. On the other hand, running the procedure with the input $g = a \cup \{y = \neg d\}$ instead of a can only lead to earlier conflicts or adding additional elements into the list *DEDUCED* due to monotony (Axiom 4) of block models. However, conflicts are impossible as g is classified by *SBCP* to be not conflicting. At the same time, since the list *DEDUCED* contains $\{y = \neg d\}$ and $\{y = d\}$ simultaneously as the result of $SBCP(g)$, the procedure must classify g to be conflicting. Thus, our supposition is not true and $y(a) = \neg d$.

Let all assignments $g = a \cup \{y = \neg d\}$ be classified by $\boldsymbol{A}$ as conflicting. Let us show that y must be constantly equal to d. Suppose, there exists a full assignment a to the inputs of $\boldsymbol{S}$ such that $y(a) = \neg d$. Then we can show that $a \Rightarrow \{y = \neg d\}$ in the same way as in the beginning of the previous paragraph. As $a \Rightarrow \{y = \neg d\}$, due to Axiom 4 (on monotony) $a \cup \{y = \neg d\}$ can not be classified as conflicting. Hence, y is constantly equal to d. ⊗

Lemma 23. Let $\boldsymbol{M}$ be a model of a normal block $\boldsymbol{B}$. The observable CNF $\boldsymbol{C}^{\cdot\cdot}$ of the model $\boldsymbol{M}$ is a model recognizing maximal conflicts under *CNF-BCP*.

Proof: Let g be a complete assignment to the pin variables of $\boldsymbol{B}$ which falsifies the permission function $\boldsymbol{f}$ of $\boldsymbol{B}$. Let a be a full assignment to the inputs of $\boldsymbol{B}$ where $a \subseteq g$. Due to Axiom 6 the assignment a implies a full assignment b to the outputs of $\boldsymbol{B}$. By lemma 1 the assignment g must contain at least one

elementary assignment $\neg b_i$ such that b_i is contained in b. Hence, $a \cup \neg b_i \subseteq g$. As $a \Rightarrow b_i$ for $\boldsymbol{M}$, there is a clause $\boldsymbol{c}^{¨} \in \boldsymbol{C}^{¨}$ which represents the assignment $a \cup \neg b_i$ or is covered by such a clause. Any case $a^{¨} \subseteq a \cup \neg b_i \subseteq g$ and $a^{¨}$ is conflicting for $\boldsymbol{C}^{¨}$ under *CNF-BCP* where $a^{¨}$ is represented by $\boldsymbol{c}^{¨}$. Due to Axiom 4 (on monotony) g is conflicting for $\boldsymbol{C}^{¨}$ under *CNF-BCP*. ⊗

Lemma 24. The observable CNF $\boldsymbol{C}^{¨}$ of the model $\boldsymbol{M}$ represents its permission function $\boldsymbol{f}$.

Proof: As each clause of $\boldsymbol{C}^{¨}$ is an implicate of $\boldsymbol{f}$, $(\boldsymbol{f} \rightarrow \boldsymbol{C}^{¨}) = 1$. As $\boldsymbol{C}^{¨}$ is a model recognizing maximal conflicts (by Lemma 23), $(\boldsymbol{C}^{¨} \rightarrow \boldsymbol{f}) = 1$. Thus, $\boldsymbol{C}^{¨} = \boldsymbol{f}$. ⊗

Lemma 25. Let an output y of a system $\boldsymbol{S}$ be tested to be constantly equal to d, and let $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ be its extended permission function where $\boldsymbol{x}$, $\boldsymbol{z}$, and $\boldsymbol{y}$ are the vectors of input, internal, and output variables of $\boldsymbol{S}$ accordingly. Then $y(\boldsymbol{x}) = d$ iff $\boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = 0$.

Proof: Let $\boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$ be the permission function of $\boldsymbol{S}$. Firstly, we show that $y(\boldsymbol{x}) = d$ iff $\boldsymbol{f}_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y}) = 0$.

⇐: Let $\boldsymbol{f}_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y}) = 0$. Suppose there exists an assignment a to the variables $\boldsymbol{x}$ such that $y(a) = \neg d$. Let $\boldsymbol{y}(a) = b$ where b is an assignment to the variables $\boldsymbol{y}$. We have $\{y = \neg d\} \in b$ and $\boldsymbol{f}(a,b) = 1$. Hence, $\boldsymbol{f}_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y}) = 1$.

⇒: Let $y(\boldsymbol{x}) = d$. Suppose there exists an assignment (a,b) to the variables $(\boldsymbol{x},\boldsymbol{y})$ such that $\boldsymbol{f}_{y=\varnothing d}(a,b) = 1$. As $\{y = \neg d\} \in b$, $y(a) = \neg d$. Thus, we have proven that $y(\boldsymbol{x}) = d$ iff $\boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y}) = 0$.

Due to Theorem 21 $\exists \boldsymbol{z}\, \boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \boldsymbol{f}(\boldsymbol{x},\boldsymbol{y})$. As $y \notin \boldsymbol{z}$, $\exists \boldsymbol{z}\, \boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \boldsymbol{f}_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y})$. Thus, $y(\boldsymbol{x}) = d$ iff $\exists \boldsymbol{z} \boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = 0$. Now, we show $\exists \boldsymbol{z} \boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = 0$ iff $\boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = 0$.

⇐: Let $\boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = 0$. Then $\boldsymbol{f}^*_{\boldsymbol{y}=\varnothing d,\, \boldsymbol{z}=g}(\boldsymbol{x},\boldsymbol{y}) = 0$ for all $g \in 2^{\boldsymbol{z}}$. As $\exists \boldsymbol{z}\, \boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \vee \boldsymbol{f}^*_{\boldsymbol{z}=g}$ $(g \in 2^{\boldsymbol{z}})$ by formula (1) and $y \notin \boldsymbol{z}$,

$$\exists \boldsymbol{z}\, \boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \vee \boldsymbol{f}^*_{y=\varnothing d,\, \boldsymbol{z}=g}\ (g \in 2^{\boldsymbol{z}}) \qquad (2)$$

Thus $\exists \boldsymbol{z}\, \boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = 0$.

⇒: Let $\exists \boldsymbol{z} \boldsymbol{f}^*_{y=\varnothing d}(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = 0$. Suppose there exists a complete assignment h to the variables $(\boldsymbol{x},\boldsymbol{z},\boldsymbol{y})$ such that $\{y = \neg d\} \in h$ and $\boldsymbol{f}^*_{y=\varnothing d}(h) = 1$. Let $h = (a,b,g)$ where $\boldsymbol{x}=a$, $\boldsymbol{y} = b$, $\boldsymbol{z} = g$. Then $\boldsymbol{f}^*_{y=\varnothing d,\, \boldsymbol{z}=g}(a,b) = 1$. Now, due to (2) $\exists \boldsymbol{z}\, \boldsymbol{f}^*_{y=\varnothing d}(a,b,g) = 1$. ⊗

Lemma 26. Let $g(\boldsymbol{x})$ be a Boolean function and let $y \in \boldsymbol{x}$. Then $g_{y=d}(\boldsymbol{x}) = 0$ iff $g(\boldsymbol{x}) \wedge y^d = 0$.

Proof:

⇒: Let $g_{y=d}(\boldsymbol{x}) = 0$. Then $g(\boldsymbol{x})$ takes the value 0, if $y = d$. On the other hand, y^d takes the value 0, if $y = \neg d$. Thus, any case $g(\boldsymbol{x}) \wedge y^d = 0$.

⇐: Let $g(\boldsymbol{x}) \wedge y^d = 0$. Then $g(\boldsymbol{x}) = 0$, if $y^d = 1$ (i.e. $y = d$). Thus, $g_{y=d}(\boldsymbol{x}) = 0$. ⊗

Proof of Theorem 25: Due to lemmas 25 and 26 $y(\boldsymbol{x}) = d$ iff $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) \wedge y^{\varnothing d} = 0$. Now, we show that $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \boldsymbol{C}_{\boldsymbol{S}}$. According to its definition, $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ is equal to the conjunction of permission functions of the normal blocks of $\boldsymbol{S}$, as adding block-constraints does not change $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ due to the fitting axiom. By Lemma 24 the observable CNF of a normal block represents its permission function. Thus, $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ is equal to the conjunction $\boldsymbol{C}_{\boldsymbol{O}}$ of the observable CNFs of the system's normal blocks. Now, we show that adding conjunctively the observable CNF $\boldsymbol{C}^{¨}$ of a block-constraint $\boldsymbol{B}$ of $\boldsymbol{S}$ into $\boldsymbol{C}_{\boldsymbol{O}}$ does not functionally change the latter, i.e. $\boldsymbol{C}_{\boldsymbol{O}} \wedge \boldsymbol{C}^{¨} = \boldsymbol{C}_{\boldsymbol{O}}$. Indeed, by definition of $\boldsymbol{C}^{¨}$, $(\boldsymbol{f} \rightarrow \boldsymbol{C}^{¨}) = 1$ where $\boldsymbol{f}$ is the permission function of $\boldsymbol{B}$. At the same time $(\boldsymbol{f}^* \rightarrow \boldsymbol{f}) = 1$ by the fitting axiom. As $\boldsymbol{f}^*(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z}) = \boldsymbol{C}_{\boldsymbol{O}}$, we finally have $(\boldsymbol{C}_{\boldsymbol{O}} \rightarrow \boldsymbol{C}^{¨}) = 1$ and $\boldsymbol{C}_{\boldsymbol{O}} \wedge \boldsymbol{C}^{¨} = \boldsymbol{C}_{\boldsymbol{O}}$. ⊗

Proof of Theorem 26: Adding a block-constraint $\boldsymbol{B}$ to a system $\boldsymbol{S}$ can have an influence on running *SYSTEM-BCP* as well as *FORCED-SYSTEM-BCP*. However, if $\boldsymbol{B}$ produces a value for a variable of $\boldsymbol{S}$, no value from the assignment affecting $\boldsymbol{B}$ at that moment can be changed due to Axiom 3. Note, that *SYSTEM-BCP* and *FORCED-SYSTEM-BCP* accumulate assignments to variables in the list *DEDUCED*. On the one hand, the block $\boldsymbol{B}$ can not change assignments from this list but can only add some additional assignments or produce additional conflicts. On the other hand, when being affected by an assignment from the list *DEDUCED*, any block $\boldsymbol{B}¢$ of $\boldsymbol{S}$ can produce only additional implications or fix additional conflicts after adding the block $\boldsymbol{B}$, as the model of $\boldsymbol{B}¢$ satisfies Axiom 4 (on monotony). Thus, an assignment classified by $\boldsymbol{S}$ as conflicting or implying before adding $\boldsymbol{B}$ will be classified as conflicting or implying afterwards. (Note, an implying assignment can be classified as conflicting after adding $\boldsymbol{B}$, however this does not decrease the value of implicativity) ⊗.

Proof of Theorem 27: Let $\boldsymbol{C}^{\cdot}$ be the characteristic CNF of the permission function $\boldsymbol{f}$ of the system $\boldsymbol{S}$. Adding to $\boldsymbol{S}$ a block-constraint with model $\boldsymbol{C}^{\cdot}$ under CNF-BCP increases its implicativity up to maximum, because $\boldsymbol{C}^{\cdot}$ under CNF-BCP is itself a model with maximal implicativity for any block with the permission function $\boldsymbol{f}$ (due to Theorem 3). ⊗.

Proof of Theorem 29: Follows from definitions. ⊗

Lemma 27. The *BDD-BCP*-procedure satisfies Axiom 5 (of our axiomatic system).

Proof: It follows from Theorem 29 that a partial assignment a is represented by an implicate $\boldsymbol{c}$ of the function f represented by a BDD $\boldsymbol{D}$ *iff* any path leading from the source node f to the sink node 1 in $\boldsymbol{D}_a$ contains an odd number of complemented edges. Thus, if the *BDD-BCP*-procedure classifies an assignment $\boldsymbol{a}$ to be conflicting, then $\boldsymbol{a}$ is represented by an implicate $\boldsymbol{c}$ of $\boldsymbol{f}$, and if the *BDD-BCP*-procedure classifies an assignment a to imply an assignment b, then for each elementary assignment $b_i \in b$, the clause representing the elementary implication $a \Rightarrow b_i$ is an implicate of the function $\boldsymbol{f}$ represented by $\boldsymbol{D}$. ⊗

Lemma 28. The *BDD-BCP*-procedure correctly simulates the Boolean vector function $y(\boldsymbol{x})$ implemented by the block $\boldsymbol{B}$ using BDD $\boldsymbol{D}$ as a model (Axiom 6).

Proof: Let the block $\boldsymbol{B}$ produce the complete assignment b to its output variables under a complete assignment a to its input variables. Then for any elementary assignment $b_i \in b$ the clause representing the assignment $a \cup \neg b_i$ is an implicate of the permission function $\boldsymbol{f}$ of the block. Due to Theorem 29 the *BDD-BCP*-procedure recognizes the assignment a to imply b_i. ⊗

Lemma 29. The *BDD-BCP*-procedure satisfies Axiom 4 (on monotony).

Proof: Let a be classified by *BDD-BCP* as conflicting. Let $a \subset a\ell$. As $\boldsymbol{D}_{a\ell}$ is a subgraph of $\boldsymbol{D}_a$ and the set of paths leading from the source node f of $\boldsymbol{D}_{a\ell}$ to its sink node 1 is a subset of analogical set in $\boldsymbol{D}_a$, $a\ell$ is classified by *BDD-BCP* as conflicting. The monotone classification of implying assignments (as it is needed for Axiom 4) is proven analogically. ⊗

Lemma 30. BDD is a consistent model under the *BDD-BCP*-procedure.

Proof: Follows from the definition of *BDD-BCP*. ⊗

Lemma 31. BDD is recognizing maximal conflicts under the *BDD-BCP*-procedure.

Proof: Follows from the definition of *BDD-BCP*. ⊗

Proof of Theorem 30: As *BDD-BCP*(a) does not change any elementary assignment to a variable presented in a, it satisfies Axiom 3. Due to Lemma 27 through 29, BDD is a model under the *BDD-BCP*-procedure. According to Lemmas 30, 31, and Theorem 9, BDD has maximal implicativity under *BDD-BCP*. Thus, due to Lemma 31 and Theorem 16, BDD has maximal strong implicativity under *BDD-BCP*. ⊗

Proof of theorem 31: Let output y of the block $\boldsymbol{B_1}$ feed the block $\boldsymbol{B_2}$. Since the C^1 and C^2 contain all prime implicates of the permission functions of blocks $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$, the only possibility for two clauses of the CNF $C^1 \wedge C^2$ to be resolved is that one of the clauses belongs to the CNF C^1 (let it be $\boldsymbol{c}^1$) and the other to C^2 (let it be $\boldsymbol{c}^2$). For the sake of distinctness, let $\boldsymbol{c}^1 = \boldsymbol{d}^1 \vee y$, $\boldsymbol{c}^2 = \boldsymbol{d}^2 \vee \neg y$. Their resolvent is equal to $\boldsymbol{d}^1 \vee \boldsymbol{d}^2$. On the other hand, $\boldsymbol{d}^2$ must contain a literal z^e of the output variable z of the block $\boldsymbol{B_2}$ (due to Theorem 1). So, $\boldsymbol{c}^2 = \boldsymbol{d}^{2\ell} \vee \neg y \vee z^e$. Considering these clauses as implicates and performing substitution of the implicate $\neg \boldsymbol{d}^1 \Rightarrow y$ into the implicate $\neg \boldsymbol{d}^{2\ell} \wedge y \Rightarrow z^e$ we obtain the implicate $\neg \boldsymbol{d}^{2\ell} \wedge \neg \boldsymbol{d}^1 \Rightarrow z^e$ that represents the same clause $\boldsymbol{d}^1 \vee \boldsymbol{d}^2$ as the resolvent above. Because both methods resolve or substitute the same pairs of clauses or implications representing the clauses, we come to the correctness of the theorem. ⊗

Proof of Theorem 32: By Theorem 21 the permission function $\boldsymbol{f}$ of the system $\boldsymbol{S}$ is equal to the extended permission function $\boldsymbol{f^*} = C^1 \wedge C^2$ existentially quantified by the variable y: $\boldsymbol{f} = \exists y\,(C^1 \wedge C^2)$. By formula (1)

$$\exists y\,(C^1 \wedge C^2) = (C^1\,(y = 1) \wedge C^2(y = 1)) \vee (C^1\,(y = 0) \wedge C^2(y = 0))\,.$$

Take into account:

$$C^1\,(y = 1) = C^1{}_{\varnothing y},$$

$C^1(y=0) = C^1_y$,
$C^2(y=1) = C^2_{\varnothing y} \wedge C^2_{\text{-}y}$,
$C^2(y=0) = C^2_y \wedge C^2_{\text{-}y}$,

Thus,

$$\begin{aligned} f &= (C^1_{\varnothing y} \wedge C^2_{\varnothing y} \wedge C^2_{\text{-}y}) \vee (C^1_y \wedge C^2_y \wedge C^2_{\text{-}y}) \\ &= ((C^1_{\varnothing y} \wedge C^2_{\varnothing y}) \vee (C^1_y \wedge C^2_y)) \wedge C^2_{\text{-}y} \\ &= (C^1_{\varnothing y} \vee C^1_y) \wedge (C^1_{\varnothing y} \vee C^2_y) \wedge (C^2_{\varnothing y} \vee C^1_y) \wedge (C^2_{\varnothing y} \vee C^2_y) \wedge C^2_{\text{-}y}. \end{aligned}$$

Now we show that $C^1_{\varnothing y} \vee C^1_y = 1$. Suppose, there exists an assignment a such that $C^1_{\varnothing y}(a) \vee C^1_y(a) = 0$. Then there are clauses $\boldsymbol{c_1} \in C^1_{\varnothing y}$ and $\boldsymbol{c_2} \in C^1_y$ such that the clauses $\boldsymbol{c_1} \vee \neg y$ and $\boldsymbol{c_2} \vee y$ can be resolved and their resolvent $\boldsymbol{c} = \boldsymbol{c_1} \vee \boldsymbol{c_2}$ is an implicate of the permission function $\boldsymbol{f_1}$ of the normal block $\boldsymbol{B_1}$. However, this is impossible due to Theorem 1 as $\boldsymbol{c}$ does not contain the output variable y of the block.

Now we show that $C^2_{\text{-}y} \Rightarrow (C^2_{\varnothing y} \vee C^2_y)$. Consider an assignment a such that $C^2_{\varnothing y}(a) \vee C^2_y(a) = 0$. Then there are clauses $\boldsymbol{c_1} \in C^2_{\varnothing y}$ and $\boldsymbol{c_2} \in C^2_y$ such that the clauses $\boldsymbol{c_1} \vee \neg y$ and $\boldsymbol{c_2} \vee y$ can be resolved and their resolvent $\boldsymbol{c} = \boldsymbol{c_1} \vee \boldsymbol{c_2}$ is an implicate of the permission function $\boldsymbol{f_2}$ of the block $\boldsymbol{B_2}$. Since the set $C^2_{\text{-}y}$ contains all such prime implicates of the function $\boldsymbol{f_2}$ that do not contain the variable y, there is a prime implicate $\boldsymbol{c}'$ covered by $\boldsymbol{c}$ in $C^2_{\text{-}y}$. Thus, $\boldsymbol{c}'(a) = 0$. Hence, $C^2_{\text{-}y}(a) = 0$. ⊗

Proof of Theorem 33: Due to Theorem 31 it suffices to consider the first two procedures. Let the output y of $\boldsymbol{B_1}$ feed $\boldsymbol{B_2}$. Then and $C^1 = (C^1_y \vee y) \wedge (C^1_{\varnothing y} \vee \varnothing y)$ and $C^2 = (C^2_y \vee y) \wedge (C^2_{\varnothing y} \vee \varnothing y) \wedge C^2_{\text{-}y}$ by formula (3) and Theorem 1. The resolution based method resolves all clauses represented by the formula $(C^1_y \vee y)$ with all the clauses represented by the formula $(C^2_{\varnothing y} \vee \varnothing y)$, and the resolvents are represented by the formula $(C^1_y \vee C^2_{\varnothing y})$. The method also resolves all the clauses represented by the formula $(C^1_{\varnothing y} \vee \varnothing y)$ with all the clauses represented by the formula $(C^2_y \vee y)$, and the resolvents are represented by the formula $(C^1_{\varnothing y} \vee C^2_y)$. No other clauses can be resolved in $C^1 \wedge C^2$ because y is the only common variable of $\boldsymbol{B_1}$, and $\boldsymbol{B_2}$, and because C^1 and C^2 consist of all prime implicates for the permission functions of the blocks $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$. Thus, the resulting CNF of the resolvent based method can be obtained from the formula $(C^1_y \vee C^2_{\varnothing y}) \wedge (C^1_{\varnothing y} \vee C^2_y) \wedge C^2_{\text{-}y}$ after performing logical addition (the operations "∨") over pairs of CNFs C^1_y and $C^2_{\varnothing y}$, and $C^1_{\varnothing y}$ and C^2_y. By Theorem 32 this CNF is the same as the one produced by the two-block quantification. ⊗

Lemma 32. Let $f(\boldsymbol{x})$ be a Boolean function where $\boldsymbol{x} = (x_1,\ldots, x_i,\ldots, x_n)$ and $\boldsymbol{c}$ be a clause depending on some variables from $\boldsymbol{x}\setminus\{x_i\}$. If $\boldsymbol{c}$ is an implicate of the function $\exists x_i\, f(\boldsymbol{x})$, then $\boldsymbol{c}$ is an implicate of the function $f(\boldsymbol{x})$.

Proof. Let a be the partial assignment represented by $\boldsymbol{c}$. Since $\boldsymbol{c}$ is an implicate of $\exists x_i\, f(\boldsymbol{x})$, $\exists x_i\, f(a) = 0$. As $\exists x_i\, f(\boldsymbol{x}) = f(x_1,\ldots, x_i = 0,\ldots, x_n) \vee f(x_1,\ldots, x_i = 1,\ldots, x_n)$, $f(a, x_i = 0) = 0$ and $f(a, x_i = 1) = 0$. Thus, the clause c representing a is an implicate of $f(\boldsymbol{x})$. ⊗

Lemma 33. Let $f(\boldsymbol{x})$ be a Boolean function, and let $f(\boldsymbol{x}) = g(\boldsymbol{u},y) \wedge h(\boldsymbol{v},y)$, where $\boldsymbol{x} = (\boldsymbol{u},\boldsymbol{v},y)$, $\boldsymbol{u} \cap \boldsymbol{v} = \varnothing$, and where y is a Boolean variable not contained in $\boldsymbol{u} \cup \boldsymbol{v}$. Let $\boldsymbol{c_1}(\boldsymbol{u}') \vee \boldsymbol{c_2}(\boldsymbol{v}')$ be an implicate of $f(\boldsymbol{x})$, where $\boldsymbol{u}' \in \boldsymbol{u}$ and $\boldsymbol{v}' \in \boldsymbol{v}$. Then $\boldsymbol{c_1}(\boldsymbol{u}')$ is an implicate of $g(\boldsymbol{u},y)$, or $\boldsymbol{c_2}(\boldsymbol{v}')$ is an implicate of $h(\boldsymbol{v},y)$, or there exists a literal y^d of y such that $\boldsymbol{c_1}(\boldsymbol{u}') \vee y^d$ is an implicate of $g(\boldsymbol{u},y)$ and $\boldsymbol{c_2}(\boldsymbol{v}') \vee y^{\varnothing d}$ is an implicate of $h(\boldsymbol{v},y)$.

Proof. Let $a' \cup b'$ be the assignment represented by the clause $\boldsymbol{c_1}(\boldsymbol{u}') \vee \boldsymbol{c_2}(\boldsymbol{v}')$, and $\boldsymbol{c_1}(\boldsymbol{u}')$ represents a', and $\boldsymbol{c_2}(\boldsymbol{v}')$ represents b'. If each complete assignment a, containing a', to the variables of the function $g(\boldsymbol{u},y)$ dissatisfies the latter, then $\boldsymbol{c_1}(\boldsymbol{u}')$ is an implicate of $g(\boldsymbol{u},y)$. If each complete assignment b, containing b', to the variables of the function $h(\boldsymbol{v},y)$ dissatisfies the latter, then $\boldsymbol{c_2}(\boldsymbol{v}')$ is an implicate of $h(\boldsymbol{v},y)$.

Now we consider the case when there is an assignment a^* to the variables of the vector $(\boldsymbol{u},y)$ such that $a' \in a^*$ and $g(a^*) = 1$, and there is an assignment b^* to the variables of the vector $(\boldsymbol{v},y)$ such that $b' \in b^*$ and $h(b^*) = 1$. Since $\boldsymbol{c_1}(a') = 0$, $\boldsymbol{c_2}(b') = 0$, and $\boldsymbol{c_1}(\boldsymbol{u}') \vee \boldsymbol{c_2}(\boldsymbol{v}')$ is an implicate of $f(\boldsymbol{x}) = g(\boldsymbol{u},y) \wedge h(\boldsymbol{v},y)$, any complete assignment g, containing $a' \cup b'$, to variables of $f(\boldsymbol{x})$ dissatisfies the latter. Thus, $g(g) = 0$ or $h(g) = 0$.

Let $y = d$ be in a^* ($d \in \{0,1\}$). Consider a complete assignment g, containing a^*, to variables of $f(\boldsymbol{x})$. Since $g(a^*) = 1$, $g(g) = 1$. Thus $h(g) = 0$. Hence, $h(b', y = d) = 0$, because h depends only of variables $(\boldsymbol{v},y)$ assigned in $(b', y = d)$. Thus, $\boldsymbol{c_2}(\boldsymbol{v}') \vee y^{\varnothing d}$ is an implicate of $h(\boldsymbol{v},y)$. As a consequence we have that the assignment b^*, dissatisfying h, must contain the elementary assignment $y = \neg d$ (recall that

b^* contains b ¢ which represents $\boldsymbol{c_2}(\boldsymbol{v}¢)$). Now by considering a complete assignment g, containing b^*, to variables of $f(\boldsymbol{x})$, we can deduce that $\boldsymbol{c_1}(\boldsymbol{u}¢) \vee y^d$ is an implicate of $g(\boldsymbol{u},y)$ (in the same way as for a^* we have proven that $\boldsymbol{c_2}(\boldsymbol{v}¢) \vee y^{Ød}$ is an implicate of $h(\boldsymbol{v},y)$). ⊗

Lemma 34. Let $f(\boldsymbol{x})$ be a Boolean function, and let $f(\boldsymbol{x}) = g(\boldsymbol{u},y) \wedge h(\boldsymbol{v},y)$, where $\boldsymbol{x} = (\boldsymbol{u},\boldsymbol{v},y)$, $\boldsymbol{u} \cap \boldsymbol{v} = \varnothing$, and where y is a Boolean variable not contained in $\boldsymbol{u} \cup \boldsymbol{v}$. Let $\boldsymbol{c_1}(\boldsymbol{u}¢) \vee \boldsymbol{c_2}(\boldsymbol{v}¢)$ be an implicate of $\exists y\, f(\boldsymbol{x})$, where $\boldsymbol{u}¢$ Í $\boldsymbol{u}$ and $\boldsymbol{v}¢$ Í $\boldsymbol{v}$. Then $\boldsymbol{c_1}(\boldsymbol{u}¢)$ is an implicate of $g(\boldsymbol{u},y)$, or $\boldsymbol{c_2}(\boldsymbol{v}¢)$ is an implicate of $h(\boldsymbol{v},y)$, or their exists a literal y^d of y such that $\boldsymbol{c_1}(\boldsymbol{u}¢) \vee y^d$ is an implicate of $g(\boldsymbol{u},y)$ and $\boldsymbol{c_2}(\boldsymbol{v}¢) \vee y^{Ød}$ is an implicate of $h(\boldsymbol{v},y)$.

Proof: follows from Lemmas 32, 33. ⊗

Proof of Theorem 34: Let $g(\boldsymbol{u},y)$ and $h(\boldsymbol{v},y)$ be the characteristic CNFs of blocks $\boldsymbol{B_1}$ and $\boldsymbol{B_2}$ respectively. Suppose there exists a prime implicate $\boldsymbol{c_1}(\boldsymbol{u}¢) \vee \boldsymbol{c_2}(\boldsymbol{v}¢)$ of $\boldsymbol{f}$ (where $\boldsymbol{u}¢$ Í $\boldsymbol{u}$ and $\boldsymbol{v}¢$ Í $\boldsymbol{v}$) that is not contained in $C(\boldsymbol{B_1}, \boldsymbol{B_2})$. Due to Lemma 34 there are three alternatives: 1) $\boldsymbol{c_1}(\boldsymbol{u}¢)$ is an implicate of $g(\boldsymbol{u},y)$, or 2) $\boldsymbol{c_2}(\boldsymbol{v}¢)$ is an implicate of $h(\boldsymbol{v},y)$, or 3) their exists a literal y^d of y such that $\boldsymbol{c_1}(\boldsymbol{u}¢) \vee y^d$ is an implicate of $g(\boldsymbol{u},y)$ and $\boldsymbol{c_2}(\boldsymbol{v}¢) \vee y^{Ød}$ is an implicate of $h(\boldsymbol{v},y)$. Let's consider them one after another.

1. This is impossible, because by Theorem 1 $\boldsymbol{c_1}(\boldsymbol{u}¢)$ must contain the output variable y of the block $\boldsymbol{B_1}$.
2. Since $\boldsymbol{c_2}(\boldsymbol{v}¢)$ is an implicate of $h(\boldsymbol{v},y)$, it covers a prime implicate $\boldsymbol{c}$ already contained in $C^2_{\text{-}y}$. As $\boldsymbol{c} \in C^2_{\text{-}y}$, $\boldsymbol{c}$ is contained in $C(\boldsymbol{B_1}, \boldsymbol{B_2})$. The only remaining case is that $\boldsymbol{c_2}(\boldsymbol{v}¢)$ is equal to $\boldsymbol{c}$.
3. In this case, (let $\boldsymbol{d} = 1$, for distinctness) $\boldsymbol{c_1}(\boldsymbol{u}¢)$ covers a clause contained in C^1_y and $\boldsymbol{c_2}(\boldsymbol{v}¢)$ covers a clause contained in $C^2_{Øy}$. The only remaining case is that $\boldsymbol{c_1}(\boldsymbol{u}¢) \vee \boldsymbol{c_2}(\boldsymbol{v}¢)$ is equal to a clause contained in $C(\boldsymbol{B_1}, \boldsymbol{B_2})$. ⊗

Proof of Theorem 35: At each step, the existential quantification procedure replaces a two-block subsystem $\{\boldsymbol{B_1}, \boldsymbol{B_2}\}$ with one block $\boldsymbol{B}$ having the CNF $C(\boldsymbol{B_1}, \boldsymbol{B_2})$ model. Due to Theorem 34 $C(\boldsymbol{B_1}, \boldsymbol{B_2})$ is the characteristic CNF of the block $\boldsymbol{B}$. Finally, the system will be replaced with one block, and the characteristic CNF of the block will be $\boldsymbol{C}^*$. ⊗

Appendix 2: Basic Notions

We list sections in which basic notions used in the paper are introduced.

Appendix 3

The basic changes to the version of the paper presented at 6th International Workshop on "Boolean Problems" 2004 (Freiberg University of Mining and Technology, Institute of Computer Science, September 23-24, pp. 103-142, 2004) are as follows.

1. We have removed Axiom 6. This axiom was necessary in the previous version, as one of the BCP procedures for a system (namely *SYSTEM_BCP*) was not able to fix conflicts itself but only with help of block models. Now this procedure can fix conflicts. It is not restrictive for practical SAT-solving, instead our theory can now be applied to the domain of simulation in which models are used for signal propagation (i.e. making implications) but not for fixing conflicts.

2. We had to add Axiom 4 on monotone classification of assignments. This axiom is naturally satisfied by realistic models. We need this axiom to prove the fundamental theorem (Theorem 26) that learning can not reduce implicativity. The proof of the theorem was omitted in the previous version, and it is not clear how to prove the theorem without this axiom.

3. We have added Section 7 in which consistent models are considered. Practical models like characteristic CNFs, BDDs, SMURFs and others are consistent. We have shown that a consistent model has maximal implicativity as well as so called maximal strong implicativity. (New Axiom 4 on monotony has been substantially used). The strong implicativity is a more refined measure for model comparison which can be used when two models have close implicativity. Thus, while constructing a good model, one can think only about its consistency, and maximal implicativity and maximal strong implicativity will be guaranteed.

4. We have proven the completeness of our system of axioms in Section 10.2 and have shown a way of reducing hierarchical SAT-solving to testing satisfiability of a CNF in Section 10.3.

Correction of Theorem 20:

Theorem 20. Let a block $\boldsymbol{B}$ has a model recognizing maximal conflicts, then procedure *CONSTRUCT_CH* delivers the characteristic CNF $\boldsymbol{C}^{\cdot}$ for $\boldsymbol{B}$. ⊗

Proof of Theorem 20. Change the last sentence of the forth paragraph as follows: Hence, due to Lemma *, the procedure must mark the node $\boldsymbol{N}$ to be conflicting and must add the clause $\boldsymbol{c}^*$ (for $a^* = a \cup \{ u = 0 \}$) to the current CNF $\boldsymbol{C}^*$.

Lemma *. Let the procedure *CONSTRUCT_CH* be applied to a block ***B*** having a model recognizing maximal conflicts, and let ***f*** be the permission function of ***B***. Let a^* be a set of decision assignments corresponding to a path ***P*** constructed by *CONSTRUCT_CH* for ***B***, and let a^* be represented by an implicate of ***f***. Let the path ***P*** lead to a node ***N***. Then *CONSTRUCT_CH* marks ***N*** as conflicting node.

Proof: As a^* is represented by an implicate of ***f*** , each complete assignment $a \in Cube(a)$ is potentially conflicting and must be classified by ***B*** as conflicting, because ***B*** is recognizing maximal conflicts. Hence, each path issued from ***N*** leads to a conflict. Due to conflict inheritance the procedure *CONSTRUCT_CH* must mark the node ***N*** as conflicting. ⊗