An Efficient Learning Procedure for Multiple Implication Checks

Yakov Novikov, Academy of Sciences (Belarus),
email: NOV@newman.bas-net.by
Evgueni Goldberg,Cadence Berkeley Labs (USA)
email: egold@cadence.com

Abstract

In the paper, we consider the problem of checking whether cubes from a set S are implicants of a DNF formula D, at the same time minimizing the overall time taken by the checks. An obvious but inefficient way of solving the problem is to perform all the checks independently. In the paper, we consider a different approach. The key idea is that when checking whether a cube C from S is an implicant of D we can deduce (learn) implicants of D that are not implicants of C. These cubes can be used in the following checks for search pruning. Experiments on random DNF formulas, DIMACS benchmarks and DNF formulas describing circuits show that the proposed learning procedure reduces the overall time taken by checks by up to two orders of magnitude.

1. Introduction

In the paper, we consider the problem of performing multiple implication checks. Given, a disjunctive normal form (DNF) *D* and a cube *C*, an implication check is to answer the question if *C* implies *D*. By multiple implication checks we mean that, such checks are done for all the cubes from a set *S* i.e. for each cube $C_i \in S$ it is checked whether C_i implies *D*. DNF *D* is further referred to as the basic one. Besides just checking if cubes from *S* are implicants our objective is also to minimize the overall time taken by all the checks.

This problem arises in synthesis [1], verification [2] of combinational circuits (e.g. false negative identification) where one often needs to check whether a vector *A* of assignments to intermediate variables is consistent. The vector specifies a cube *C* in the Boolean space of circuit variables. To see if *A* is consistent one needs to check if *C* is an implicant of a DNF *D* specifying the set of consistent assignments to circuit variables.

A common way of checking whether a cube *C* is an implicant of a DNF *D* is as follows. First, we form the cofactor D_C of *D* with respect to cube *C* and then check if D_C is a tautology [1]. (A DNF *D* is a tautology if it is equal to 1 for any value assignment to the variables of *D*.) In other words, we check if *D* turns into a tautology after making the assignments setting the literals of *C* to 1. (For example, if *C* is equal to $a\bar{b}c$ then these assignments are a=1,b=0,c=1). Each tautology check is a co-NP complete problem [4]. If all the checks are performed separately, then the more cubes are in set *S* the harder the problem of multiple implication checks becomes.

In general, one can make use of previous implication checks by adding to basic DNF *D* every cube $C_i \in S$ that has been proven to be an implicant of *D*. There are two problems in such an approach. Firstly, this kind of learning doesn't work if there are no implicants in *S*. Secondly, even if a cube $C_i \in S$ is an implicant, learning just this fact is a small share of information one can actually get during implication check.

The point is that if one uses conflict analysis [11] then when checking if C_i is an implicant it is possible to deduce implicants of *D* that are not implicants of C_i. (For tautology checking, conflict is the situation when under a set of assignments *A* a cube *C* of the original DNF *D* cannot be set to 0, so the current DNF is a tautology.) Conflict analysis is based on the observation that some assignments on a path *P* leading to a conflict are redundant. This means that one will get the same conflict for all the paths that can be obtained from *P* by flipping redundant assignments. This fact can be stored in a database as the cube specified by an irredundant subset of the set of assignments on path *P*. GRASP [9] is an example of an algorithm storing a database of conflicts to prune search tree.

Similar conflict analysis can be used for implication checks. Suppose that *C*=*abc* is a cube to be checked if it is an implicant of *D*. If we construct D_C for tautology checking, we lose information about the original formula *D*. It means that for any conflict encountered during tautology check of D_C we consider assignments a=1,b=1,c=1 to be irredundant. In our approach we rather consider implication check as examining a branch (starting with assignments a=1,b=1,c=1) of the search tree constructed when tautology checking formula *D* (not D_C). This allows one to check in each encountered conflict whether assignments a=1,b=1,c=1 are irredundant. As the result we can obtain implicants of *D* that are not implicants of *C*. In other words we can deduce implicants whose influence on other implication checks goes "beyond" cube *C* itself. To limit the number of cubes to be stored, in the current version of our algorithm only cubes whose number of literals is below a threshold are stored.

Treating implication check as examination of a branch of the search tree constructed when tautology checking *D* looks like a minor technical issue. However, it means an important paradigm shift. Instead of asking the same question about different objects (is this cofactor of *D* a tautology?) we ask different questions about the same object, which is the basic DNF *D* (is this cube an implicant of *D*?). This allows one to become increasingly knowledgeable about this object. It should be noted that the fact, that adding implicants to basic DNF *D* makes it easier to perform next implication checks, is obvious. The point however is that we get implicants of *D* without any extra effort just as a by-product of implication checks.

In [5],[10] a similar problem has been addressed. Rephrased in tautology checking language it can be formulated as the problem of multiple tautology checks performed on a set of DNFs sharing many cubes. In [10] a technique of storing "pervasive" cubes was proposed. The difference between our approach and this technique is explained in detail in section 5.

Our experiments on random and circuit DNFs show that the proposed learning procedure reduces the overall time taken by the checks by up to 2 orders of magnitude.

2. Unit cube rule and BCP

Suppose that we perform a tautology check on DNF *D* and the latter contains a unit cube i.e. a cube containing one literal, say cube $C=a$. Then we can immediately conclude that no solution exists with $a=1$ (by solution we mean an assignment proving that *D* is a non-tautology i.e. an assignment under which $D=0$). So from the unit cube *C* we can deduce that the value taken by variable *a* in any solution (if *D* is a non-tautology) is equal to 0. This way of deducing values is called the unit cube rule. Now we can simplify *D* by making assignment $a=0$ and discarding all cubes having literal *a* and removing literal $\overline{a}$ from all cubes that have it.

After applying the unit cube rule once, new unit cubes may appear in the current DNF so the rule can be applied again. The procedure of iterative application of the unit cube rule is called Boolean Constraint Propagation (BCP). BCP stops when

- there are no unit cubes in the current DNF;
- a solution is found;
- a conflict is encountered.

When performing a tautology check on a DNF *D*, conflict means the situation when all literals are removed from a cube *C* of *D* under current assignment (we will say that a conflict on cube C is encountered). So no solution can be obtained by branching on unassigned yet variables.

3. Conflict analysis

Conflict analysis is based on the following observation. Let *A* be a set of assignments that after performing BCP lead to a conflict on a cube *C*. It means that under assignment *A* followed by BCP all literals of *C* are removed and it can't be set to 0. It may happen that even after removing some subset *A'* of assignments from *A*, the rest of assignments lead to the same conflict on *C* [11]. So assignments *A'* are redundant with respect to the conflict. This observation allows one to prune some paths different from the current one. Suppose for example, that having made assignments $A=\{a=1,b=0, v=1,m=1,s=0,d=1\}$ we get a conflict. This means that cube $C = a\overline{b}vm\overline{s}d$ specified by assignments *A* is an implicant of the original DNF. Storing the cube doesn't make sense because the algorithm will never explore a path containing all the assignments from *A*. Suppose, however, that conflict analysis shows that assignments $a=1,s=0$ are redundant. Then we can claim that cube $C^* = \overline{b}vmd$ strictly containing *C* is an implicant of *D* too. Storing cube C^* does make sense. Suppose that $a=1$ and $s=0$ are first assignments to variables *a* and *s* i.e. assignments $a=0$ and $s=1$ are yet to be examined. Then storing C^* will help to prune branches corresponding to the sets of assignments obtained from *A* by flipping values of variable *a* and/or variable *s*.

It is convenient to use a special term for a set of assignments leading to a conflict. We will call it conflict recipe. As it was mentioned above, conflict recipes can be redundant, which means that the same conflict can be described by different recipes. Cube $C^* = \overline{b}vmd$ specifies an irredundant recipe $\{b=0,v=1,m=1,d=1\}$ of the same conflict as recipe *A* above. Moreover, the same conflict can have different irredundant recipes. Recipes differ in their quality and a number of techniques have been proposed for recipe optimization. In [7] a procedure for reducing recipe size by replacing conscious (i.e. decision) assignments of the recipe with deduced ones is introduced. In [9] a technique for splitting a recipe into two shorter ones is suggested. These techniques are used in our experiments described in section 6.

4. Problem formulation and algorithm description

The problem of multiple implication checks is formulated as follows. Given a DNF formula *D* (called basic DNF) and a set of cubes $S=\{C_1,\ldots,C_n\}$, check for every cube C_i ,$i=1,..,n$ if it is an implicant of *D* minimizing the overall time taken by all *n* checks. C_i is an implicant of *D* if for any assignment for which C_i is equal to 1, *D* is equal to 1 as well.

Our algorithm performs implication checks one by one, with no particular order. For the sake of simplicity, let us assume that cubes are checked according to their numbers i.e. C_1 is the first to check and C_n is the last. (Though one has every reason to believe that the order in which cubes of S are processed may dramatically affect algorithm performance, finding a good ordering is not the focus of the paper). Let C_i be the next cube to check. The key idea of our algorithm is to store (i.e. to add to D) implicants of D obtained when performing implication check on C_i. The reason for keeping such implicants is that conflict analysis allows one to deduce implicants of D which are not implicants of C_i. These implicants may be useful for next implication checks. To limit the number of implicants of D we keep only the ones whose number of literals is less than a threshold are stored.

Let us first describe the difference between our and traditional approaches. Assume for the sake of clarity that C_i = abc is the next cube to check. A common way of performing implication check on C_i is to compute the cofactor D_{C_i} of basic DNF D by making assignments a=1,b=1,c=1 and then perform a tautology check on D_{C_i}. The drawback of such approach is that we assume that all three assignments above "contribute" to any conflict encountered when performing tautology check on D_{C_i}. So all implicants of D_{C_i} that are deduced are implicants of C_i (because we have to add all three literals a,b,c to any implicant deduced when performing tautology check on D_{C_i}).

The key point of our approach is that instead of checking if D_{C_i} is a tautology we examine the branch a=1,b=1,c=1 of the tree constructed when checking if basic DNF D is a tautology. If no solution is found in this branch, then C_i is an implicant of D, otherwise it is not. This may look like a minor trick but in fact, it means an important paradigm shift. In a traditional approach we ask the same question (is this DNF a tautology?) about different objects, which are cofactors $D_{C_1},\ldots,D_{C_n}$. In our approach we ask different questions (is this cube an implicant of the basic DNF D?) about the same object, which is the basic DNF. Accumulating implicants of D, the algorithm becomes more and more "knowledgeable" about the object it deals with.

Let us consider in more detail how an implication check is performed. Let C_i be the cube to perform implication check on. First, the assignments setting all literals of C_i to 1 are made in D (we will refer to them as initial assignments). If, for example, C_i=abc then the set of initial assignments is a=1,b=1,c=1. This means that all cubes having at least one of the literals $\overline{a},\overline{b},\overline{c}$ are discarded and literals a,b,c are removed from all the cubes having them. If necessary, BCP is applied. After making the initial assignments the following three situations are possible.

1) A conflict if found. It means that the cube specified by the initial assignments (in our example it is cube abc) is an implicant of D.

2) A solution is found i.e. D is set to 0. Then C_i is not an implicant of D.

3) If neither of the cases above occurs then the algorithm starts branching trying to find an assignment setting all the cubes of D to 0. If a conflict is encountered, a recipe of the conflict is constructed and added to D. Nonchronological backtracking is performed [9]. If a solution is found, which proves that C_i is not an implicant of D, the implication check stops. If algorithm backtracks to one of the initial assignments, which means that C_i is an implicant of D, the implication check stops.

When an implication check of cube C_i is completed, all found implicants added to D during of C_i checking and containing more literals then the threshold value are deleted from D. So D accumulates only short implicants which have more chances to be useful in the implication checks to follow.

Let us illustrate the method by the following example. Let $S = ab \vee ag \vee b\overline{g}f$ be the DNF specifying the set of cubes to check if they are implicants of a basic DNF $D = ac \vee b\overline{g}h \vee \overline{c}gh \vee \overline{h}f \vee \overline{h}\overline{f}$. Let cube ab be the first to be checked. After making the initial assignments a=1 and b=1 we get DNF $c \vee \overline{g}h \vee \overline{c}gh \vee \overline{h}f \vee \overline{h}\overline{f}$. It contains single unit cube c from which we deduce c=0. By making the assignment we reduce the DNF to $\overline{g}h \vee gh \vee \overline{h}f \vee \overline{h}\overline{f}$. Since BCP stops here, we need to choose a variable to branch on. Suppose that we branch on variable g and branch g=0 is examined first. Substituting g for 0 we get DNF $h \vee \overline{h}f \vee \overline{h}\overline{f}$. From the unit cube we deduce h=0. After assigning h=0 we get DNF $f \vee \overline{f}$. Now deducing either assignment f=0 or f=1 leads to a conflict on cube $\overline{h}f$ or cube $\overline{h}\overline{f}$ of the basic DNF D.

The conflict depends on chosen assignments b=1 and g=0 (and does not depend on assignment a=1). Indeed, after making assignments b=1 and g=0 we get DNF $D = ac \vee h \vee \overline{h}f \vee \overline{h}\overline{f}$. After deducing h=0 and applying BCP we get the same conflict on cube $\overline{h}f$ or cube $\overline{h}\overline{f}$ as before. So the set of assignments {b=1,g=0} can be considered as a recipe of the conflict. The recipe specifies an implicant $b\overline{g}$ of the basic DNF which is not an implicant of cube ab for which implication check is performed. Now we need to backtrack and flip the assignment to the last chosen variable responsible for the conflict i.e. variable g. Assignment g=1 leads to the same DNF $h \vee \overline{h}f \vee \overline{h}\overline{f}$ that was considered in the branch g=0.

After performing BCP we obtain a conflict with recipe $\{a=1, g=1\}$. Again this recipe specifies an implicant ag of the basic DNF which is not an implicant of ab.

Now we need to backtrack to variable b which is an initial variable. This means that implication check is over and cube ab is an implicant of D.

Suppose the threshold is equal to 2. So the two found implicants $b\bar{g}$, ag can be added to the basic DNF D. So $D = ac \vee b\bar{g}h \vee \bar{c}gh \vee \bar{h}f \vee \bar{h}\bar{f} \vee b\bar{g} \vee ag \vee ab$. Now checking ag, $b\bar{g}f$ becomes trivial because ag is already contained in D and $b\bar{g}$ contains $b\bar{g}f$ (that is $b\bar{g}f$ implies $b\bar{g}$). So after making initial assignments b=1, g=0, f=1 we get a conflict on cube $b\bar{g}$ and immediately conclude that $b\bar{g}f$ is an implicant of D.

5. Our approach versus storing pervasive cubes

In [5],[10] the problem of multiple tautology checks performed on a set of "similar" DNFs $D_1,\ldots,D_n$ was addressed. The DNFs are similar in the sense that they share a substantial set of cubes. In [10] a technique of storing "pervasive" cubes was introduced. Let D^* be the set of cubes shared by all the DNFs to be tautology checked. A cube C is pervasive if it is an implicant of DNF D^*. Since D^* is contained in every DNF D_i then obviously C is an implicant of D_i as well. So storing a pervasive cube is useful because it is an implicant of all the DNFs and can be employed in the following tautology checks.

As it was mentioned in the introduction the problem of checking if a cube C is an implicant of F reduces to the tautology check of the cofactor F_C. So checking if cubes from the set $S=\{C_1,\ldots,C_n\}$ are implicants of a DNF D reduces to multiple tautology checks performed on the set of cofactors $D_{C_1},\ldots,D_{C_n}$. Then the technique of storing pervasive cubes can be applied to reduce the overall time of tautology checks. However there are at least three reasons why our approach is superior when applied to the problem of multiple implication checks.

1) Our approach is applicable even when the set S of cubes to perform implication check on is not known beforehand. In this case the set of cofactors $D_{C_1},\ldots,D_{C_n}$ is not known beforehand either and so the set D^* of shared cubes cannot be computed.
2) Even if set S is specified before starting implication checks it is quite possible that the set D^* of cubes shared by the cofactor DNFs is very small. (It can be actually empty if in cubes of S either literal of each variable of DNF D appears at least once because then for each cube C of D there is a cofactor DNF from which C is discarded.)

Table 1. Results on random functions

Basic DNF D			Set of cubes S		Thresh. F=0,1,2		Threshold F=3			Threshold F=4			Threshold F=5			Threshold F=6		
	V	*C*	*L*	*I*	*Tree*	*T*	*Tree*	*%T*	*New*	*Tree*	*%T*	*New*	*Tree*	*%T*	*New*	*Tree*	*%T*	*New*
1	100	410	1000	824	183	200	152	93	126	45	31	965	29	24	1514	15	16	1863
			2000	1641	183	407	96	75	119	21	23	100	4	14	1	5	11	7
			3000	2446	182	610	78	66	71	16	19	11	4	10	2	5	10	10
			4000	3249	183	814	43	56	256	17	17	24	4	9	0	6	10	7
			5000	4042	182	1012	12	47	148	16	16	14	4	8	2	6	9	7
2	100	410	1000	273	201	192	137	73	88	76	48	807	45	31	1906	34	26	2423
			2000	544	201	404	126	67	24	59	44	101	26	24	343	16	19	383
			3000	823	205	602	110	64	27	59	43	68	23	22	194	16	17	282
			4000	1112	196	796	110	62	33	49	41	92	20	20	157	14	16	212
			5000	1404	202	987	91	60	147	39	38	45	18	19	165	14	16	167
3	100	410	1000	449	235	263	175	79	118	57	31	997	37	20	2036	31	20	2830
			2000	912	230	524	125	69	161	27	22	317	19	15	267	12	14	311
			3000	1358	239	785	125	65	176	25	18	182	19	14	188	11	12	201
			4000	1812	228	1055	121	63	90	25	17	154	18	13	167	10	11	90
			5000	2252	288	1313	114	62	85	24	16	134	18	13	139	10	11	64

3) Storing only cubes which are implicants of D^* is unnecessarily restrictive. Indeed, suppose that set S consists just of two cubes $C_1=abc$ and $C_2=xyz$. Then the set D^* of cubes shared by the cofactors D_{C_1}, D_{C_2} consists of all the cubes of D that don't contain (positive and negative) literals of variables a,b,c,x,y,z. However it is not hard to show that using cubes of D containing, say, literal b our algorithm may obtain a cube C^* (having literal b and not having literals a,c) which is an implicant of D and is not an implicant of C_1. Cube C^* may turn to be useful when checking if C_2 is an implicant of D, for example, in branch b=1 if b is chosen as a branching variable (and in many other cases).

6. Experimental results

In our experiments we used a conflict analysis based algorithm which employs a number of known techniques like the one of using deduced assignments in conflict recipes [7] and the procedure of dominator identification [9]. However it doesn't use the concept of decision levels [9] to make search tree size computation easier. In each experiment a basic DNF D and a set S of cubes, to be checked if they are implicants, are given. After processing a cube $C_i \in S$ all the found implicants of D whose size is less than a threshold value F are added to D. (The size of a cube is equal to the number of its literals.) If F equals 0 no implicants are added to D which corresponds to performing implication checks separately. The algorithm is implemented in Visual C++ 4.0 for Windows 95. The experiments were run on a computer with AMD-230 CPU and 32 Mbytes of RAM.

In table 1 results on random DNF formulas are given. We consider three basic DNFs that are hard non-tautologies [6],[12]. Each DNF consists of 3-literal cubes and the number of cubes in it is 4.1 times the number of variables. For all three DNFs we use the same set S of 5000 cubes to perform implication checks, each cube having 3 literals. The basic DNFs are selected so as to cover all three possible cases of splitting S into implicants and non-implicants: the majority of implicants (first DNF), the majority of non-implicants (second DNF), equal shares of implicants and non-implicants (third DNF). In table 1 results after every 1000 implication checks are given.

Columns V and C specify the number of variables and cubes in the basic DNF. Columns L and I specify the number of checked cubes and the number of implicants among them. Column *Tree* gives the average tree size (the number of nodes) over every processed set of 1000 cubes. Column *New* gives the number of implicants added to D. Column T specifies the overall time (in seconds) of 1000 implication checks for threshold value F=0 (independent checks). Column %T specifies the ratio (percent) of the overall time taken by 1000 checks for a non-zero value of F and for F=0 .

It is not hard to see that for thresholds F=4,5,6 average tree size *Tree* and overall runtime decrease every 1000 cubes as does the number *New* of added implicants. For example for the first basic DNF, processing 1000 cubes is sufficient to reduce the average tree size from 182 nodes for F=0 to 4 nodes for threshold F=5 for all the next implication checks. Besides, the process of learning is nearly complete after processing first 1000 cubes, since only 5 new implicants are added to the basic DNF.

In table 2 results on some DNFs that are representatives of DIMACS suite classes [3] are given. Column Type specifies the DNF type: B denotes the basic DNF and S means that the cubes of this DNF are checked if they are implicants of B. Column N gives the total number of nodes in the search trees built for all the implication checks performed for cubes of S. Column %N specifies the ratio of the total number of nodes for a non-zero value of F and for F=0. Column %T gives the ratio of the overall time taken by all checks for a non-zero value of F and for F=0.

It should be mentioned that all the basic DNFs from table 2 are tautologies, which means that all cubes of S are implicants of B. One can see from the table that for example for threshold F=5 the average tree size is just 3.3% of the tree size for F=0 and for threshold F=4 the average total time of checks is just 2.1% of the time for F=0.

In table 3 results for DNFs from the DIMACS class Aim200 are given. Here the basic DNFs are not tautologies. Class Aim200 consists of 6 subclasses: Aim200-1-6yes, Aim200-2-0yes, Aim200-3-4yes, Aim200-6-0yes, Aim200-1-6no, Aim200-2-0no. The first four subclasses are non-tautologies and DNFs of the subclasses are used as basic DNFs B. The DNFs of the last two subclasses that are tautologies are used to represent sets of cubes S. Each of the six subclasses contains 4 DNFs. So in total we have 16 multiple implication checks problems for each pair (DNF B from a class of non-tautologies, DNF S from a class of tautologies). Table 3 gives average results for each set of 16 problems. Column I_a specifies the average number of implicants of B in set S for each set of 16 problems. Column NI_a specifies the average number of non-implicants in set S for each set of 16 problems. In contrast to columns with the same name from previous tables columns N, T, C, %N, %T, *New* are averaged over sets of 16 problems.

Table 2. Results on DIMACS representatives of different classes

Name	V	C	*Type*	Threshold F=0,1		Threshold F=2			Threshold F=3			Threshold F=4			Threshold F=5		
				N	T	%N	%T	New	%N	%T	New	%N	%T	New	%N	%T	New
Dubois20 Pret60-25	60 60	160 160	*B* *S*	40582	7	8	6.7	15	6.1	6.3	77	3.5	5.1	107	2.2	4.3	155
Pret60-25 Dubois20	60 60	160 160	*B* *S*	65220	10	100	100	0	3.6	4.8	74	1.9	2.2	144	1.9	4.5	185
Pret150-25 Dubois50	150 150	400 400	*B* *S*	1256178	197	100	100	0	0.47	0.56	174	0.43	0.64	376	0.42	0.61	393
Dubois50 Pret150-25	150 150	400 400	*B* *S*	827048	277	3.6	2.5	15	2.4	1.4	129	0.6	0.66	582	0.47	0.7	582
Dubois24 Hole8	72 72	297 192	*B* *S*	125274	23	2.1	1.7	21	3.6	4.0	70	1.7	2.7	105	1.4	2.4	173
Hole8 Dubois24	72 72	192 297	*B* *S*	53409	97	18	13	44	16	9.7	55	16	10	56	15.6	10	61
Average				187327	102	39	37	16	5.4	4.5	83	4	2.1	228	3.3	2.3	258

Table 3. Results on DIMACS class Aim200

1- set of cubes S from subclass Aim200-1-6no; 2- set of cubes S from subclass Aim200-2-0no

	Basic DNF subclass	Set of cubes S		Threshold F=0			Threshold F=1			Threshold F=2			Threshold F=3			Threshold F=4		
		I_a	NI_a	N	T	C	*%N*	*% T*	*New*	*%N*	*% T*	*New*	*%N*	*% T*	*New*	*%N*	*% T*	*New*
1	Aim200-1-6yes	279	40	43234	13	318	43	40	19	6,8	8,7	127	3,6	6,7	199	3,9	7,2	242
	Aim200-2-0yes	282	37	61589	26	397	21	17	30	4,7	5,3	136	2,7	4,2	245	2,9	4,6	289
	Aim200-3-4yes	278	41	56729	72	678	18	12	35	3,6	2,6	127	2,6	2,0	205	2,8	2,1	254
	Aim200-6-0yes	279	40	48025	97	1185	11	6,8	9	3,7	2,1	80	3,5	2,2	135	3,6	2,5	194
2	Aim200-1-6yes	347	51	53196	16	318	39	37	21	7,6	9,1	145	3,6	6,4	243	3,6	7,0	273
	Aim200-2-0yes	348	50	77050	33	397	18	16	31	4,3	4,8	148	2,6	4,1	269	2,7	3,5	301
	Aim200-3-4yes	350	48	71672	94	678	17	13	31	3,1	2,4	98	2,6	1,9	214	2,8	2,1	271
	Aim200-6-0yes	349	49	58947	119	1185	17	14	10	3,5	2,0	89	3,4	2,1	132	2,9	2,0	194
Average		314	45	58805	59	645	23	20	23	4,7	4,6	24	3,1	3,7	180	3,2	3,9	252

As one can see from table 3 for threshold $F=3$ the average tree size is only 3.1*%* of the tree size for F=0 and time is 3.7*%* and on average 180 implicants are added to basic DNFs.

In table 4 the results for checking consistency of assignments to intermediate variables of circuits are given. Given a circuit N, the basic DNF D here is a DNF specifying all consistent (observable) assignments to variables of N. To check if a set of assignments to a few variables of N is observable one must check if the cube specified by the assignments is an implicant of D. In each experiment 5000 cubes of 3-literals were randomly generated, each cube specifying a combination of assignments to 3 intermediate variables of N.

Columns V and C specify the number of variables and cubes in the basic DNF D. Column *Tree* gives the average tree size for F=0. The next 3 columns specify results for the threshold (indicated in column F') for which the greatest average tree size reduction is achieved. Column *Tree'* gives the average tree size over the last 1000 implication checks and *New'* is the total number of implicants added to D (after all 5000 checks). It should be noted that though the average tree size is computed only for the last 1000 checks, the learning process for threshold

F' is nearly complete after a fraction of the total 5000 checks which doesn't exceed a few hundred checks.

The last 3 columns give results for the threshold (specified in column *F*") for which the greatest runtime reduction is achieved. Column %*T*" specifies the ratio (percent) of the overall runtime for threshold values *F*=0 and *F*". Column *New*" gives the total number of implicants added to the basic DNF *D*.

Table 4. Results for circuit DNFs

Basic DNF *D*			*F*=0	Best tree minimization threshold			Best time minimization threshold		
Circuit	*V*	*C*	*Tree*	*F'*	*Tree'*	*New'*	*F"*	*%T"*	New"
Cht	155	697	450	2	2.6	20	5	2.6	175
Lal	187	679	119	2	2.8	43	1	2.6	12
b9	296	795	271	3	2.9	159	3	2.9	159
Example2	331	1311	436	2	2.7	190	3	1.1	206
Term	338	1791	702	5	2.9	581	5	0.9	581
Count	145	443	117	4	2.4	104	3	4.0	83
9symml	97	936	39	4	2.4	450	4	9.8	450
i3	279	557	274	2	7	137	2	4.1	137
i4	387	893	1750	2	28	68	2	2.0	61
Sct	115	513	48	4	2.3	90	2	11	24
ttt2	180	1007	278	5	2.6	453	5	2.0	453
Apex7	204	911	911	2	2.6	139	5	0.8	401
Comp	145	453	188	5	2.5	238	5	2.5	238
Cordic	229	601	158	4	2.2	286	5	1.3	220
Average	221	828	410	3.3	4.7	211	3.6	2.7	229

7. Conclusions

We consider the problem of checking if cubes $C_1,..,C_n$ are implicants of a DNF *D* and suggest an efficient procedure of "on-the-fly" learning. We introduce a technique that allows one during the implication check of C_i to deduce cubes that are implicants of *D* but not implicants of C_i . The technique gives a "cheap" and effective way of reducing the complexity of subsequent implication checks.

One of the possible directions for future research is studying how order in which cubes $C_1,..,C_n$ are processed affects the overall runtime and finding heuristics allowing one to select best orders.

References

[1] R.K.Brayton et. al. *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[2] Burch, J.R.; Singhal, V. Tight integration of combinational verification methods. *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers.* , 1998, Page(s): 570 -576

[3] *DIMACS Challenge benchmarks* in ftp://Dimacs. Rutgers.EDU/pub/challenge/sat/benchmarks/cnf.UCSC benchmarks in/pab/challenge /sat /contributed /UCSC.

[4] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, 1979.

[5] J.Kim,J.Whittemore,K.Sakallah. Improving SAT: Stack-Based Incremental Satisfiability. *Notes of the International Workshop on Logic Synthesis,IWLS-2000*.

[6] Mitchel D. et. al. Hard and easy distibutions of SAT problems. *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pp.459-465.

[7] Novikov Y. Using intermediate conflict recipes for solving satisfiability problem. *Proceedings of the Third International Conference on computer-aided design of discrete devices CADDD'99*, Minsk 1999, pp.148-153.

[8] Savoj, H.; Brayton, R.K. Observability relations and observability don't cares. *Computer- Aided Design, 1991. ICCAD-91. Digest of Technical Papers.*, 1991, pp. 518 - 521

[9] Silva J.P.M, Sakallah K.A. GRASP - a new search algorithm for satisfiability. *ICCAD-96*.

[10] Silva J.P.M, Sakallah K.A. Robust search algorithms for test pattern generation. *Fault-tolerant computing,FTCS-27*. Digest of papers,1997,pp.152-161.

[11] Stableman R.M., Susan G.J. Forward Reasoning and Dependency-Directed Backtracking. A system for Computer-Aided Circuit Analysis. *Artificial intelligence*, vol. 9, pp. 135-196, October 1977.

[12] Utkin A.A. Experimental Investigation of Satisfiability Algorithms. *Avtomatika i Vichislitelnaya Technika*, No. 6, pp.66-74, 1990 (in Russian).