Research Report

Improved Decision Heuristics for High Performance SAT Based Static Property Checking

Emmanuel Zarpas[1], Marco Roveri[2], Alessandro Cimatti[2], Klaus Winkelmann[3], Raik Brinkmann[3], Yakov Novikov[3], Ohad Shacham[1], Monica Farkash[1]

(Deliverable 3.2/1)

[1] IBM

[2] ITC-irst

[3] Infineon

TABLE OF REVISIONS

Issue	Date	Description + Reason for the Modification	Affected Sections
0.1	June 11, 2004	Creation	
0.2	June 20, 2004	Integration of first draft from each partner	
0.3	June 27, 2004	Review	
1.0	June 30, 2004	First complete version	

Abstract:

This document contains important aspects of SAT Solver performances and specification for improving this performance.

CONTENTS

1 INTRODUCTION AND BACKGROUND 5

1.1 MOTIVATION 5
1.2 TODAY'S SAT SOLVERS 5

2 BENCHMARKING OF SAT SOLVERS 7

2.1 BENCHMARKING OF ZCHAFF, BERKMIN AND SIEGE 7
2.2 COMPARISON BETWEEN ESOLVER, SIEGE AND DPSAT (INFINEON) 13
2.3 EXPERIMENTAL RESULTS AND CONCLUSIONS 15

3 THE IMPROVEMENTS OF SAT BASED BOUNDED MODEL CHECKING 19

3.1 DECISION HEURISTICS FOR TUNING SAT 19
3.2 A SIMPLE BUT EFFICIENT DISTRIBUTED SAT BMC ALGORITHM 23
3.3 CONCLUSIONS 25
3.4 THE IBM FORMAL VERIFICATION BENCHMARK 25

4 FUTURE ACTIVITY: INVESTIGATING INCREMENTALITY 28

4.1 INCREMENTAL SAT 28
4.2 BMC AND INCREMENTAL BMC 29

5 REFERENCES 34

1 Introduction and Background

Algorithms for checking the satisfiability of propositional logic are widely used in verification and analysis of hardware and software systems. The research around SAT-procedures started in the context of automated theorem proving, where SAT was identified as a simple instance of formally proving theorems, but in the last years there were many advances driven by the electronic design automation (EDA) community with their huge interest in efficiently solving large HW problems. Hence in the past years we saw a rise in the number of SAT solvers available on the market, and their efficiency in solving HW verification problems.

All SAT tools are based on search algorithms that aim at finding a satisfying assignment by variable splitting. The continuously improving results have been provided by work on good heuristics and algorithms. As proved by the past experience with SAT solvers, once found a good heuristics or another improvement for SAT solver, it can be implemented and used by all.

The purpose of this Prosyd task is to share information regarding SAT Solvers performances, and find ways to improve their performance.

1.1 Motivation

The classical NP-complete problem of boolean satisfiability (SAT) has seen much interest in areas where practical solutions to this problem enable significant practical applications. Since the development of the DPLL algorithm, this area has seen active research that generated the current state-of-the-art SAT solvers, able to handle problem instances with thousands, and in same cases even millions, of variables.

There are several SAT algorithms that have seen practical success. These algorithms are based on various principles such as resolution, search, local search and random walk, Binary Decision Diagrams, Stälmarck's algorithm, and others. We will take a look at some of SAT solvers used in EDA.

1.2 Today's SAT Solvers

The solutions offered by SAT solvers to the following major issues differentiate among them:.

Learned conflict clauses. Adding conflict clauses allows one to prune many of the branches of the search tree that are yet to be examined.

Restart. Some SAT-solvers use restarts when the SAT-solver abandons the current search tree (without completing it) and starts a new one.

Locality. Locality is an important advancement in pruning the search space. The variable score (VSIDS) and branching on variables within certain locality (BerkMin, Siege) exhibit great speedups.

Clauses. Long clauses are very harmful when they are not frequently used. Shorter clauses lead to faster BCP and quicker conflict detection.

The variation in solutions chosen for the above gives us a large variety of solvers. Choosing the most efficient solution to implement should be a result of looking at the existing solutions. There are though problems in doing so: finding the touchstone on which to test the existing solutions, the complexity in comparing the results, and finally in deciding what is efficient for each tool separately. For the purpose of this document we took a look at several important SAT solvers.

Zchaff.

Zchaff was initially released in 2001 by the Princeton University as a new implementation of the Chaff algorithm. The SAT 2002 Competition considered it the Best Complete Solver in both industrial and handmade benchmark categories. Zchaff was a pioneering solution and continues to be an important player in the field by amasses the best of Princeton university research. It is especially popular with user applications that take advantage of its version as a linkable library (e.g. NuSMV). Its main features are:

- the Two Literal Watching scheme for boolean constraint propagation,
- the Variable State Independent Decaying Sum (VSIDS) scores for decision making and locality centric decision strategy,
- non-chronological backtracking with multiple conflict analysis,
- adoption of a rapid restart policy,
- an aggressive clause database management.

BerkMin.

BerkMin (Berkeley-Minsk) [GN] is a descendent of the family of SAT-solvers that includes GRASP, SATO, Chaff. BerkMin uses the procedures of conflict analysis and non-chronological backtracking (GRASP), fast BCP (SATO), variable decaying (Chaff), and restarts. BerkMin individualizing features are visible in:

- decision making (branching variable is chosen among the free variables, whose literals are in the top unsatisfied conflict clause),
- clauses database management (chronologically ordered stack, decision to remove depends on age and activity)
- heuristics in choosing which assignment of the chosen branching variable to be examined first,
- a special algorithm to compute the activity of variables, taking into account a wider set of clauses involved in conflict making than generally used.

Siege

Siege is a SAT solver provided by the Simon Fraser University. It is known that it implements as a decision strategy VMTF and the uses the BCP techniques.

E-Solver.

E-Solver is a SAT solver developed experimentally by Infineon.

DPSat.

DPSat is Infineon's currently productive solver DPSat.

The next chapter is taking a look at benchmarking problems, and tries to compare the performance of the above solvers. The following chapters offer ideas on how to improve the performance of existing solvers.

2 Benchmarking of SAT Solvers

Modern SAT solvers are highly dependant on heuristics. Therefore, benchmarking is of prime importance in evaluating the performances of different solvers. However, making relevant benchmarking is not necessarily straightforward. We present IBM and Infineon experiments using the IBM CNF Benchmark on several SAT solvers.

2.1 Benchmarking of zChaff, BerkMin and Siege

2.1.1 Experimental results

Our experiments compared three famous SAT solvers, zChaff, BerkMin561 and siege_v4. In these experiments we use the IBM CNF Benchmark, with CNFs generated by BMC from the IBM Formal Verification Benchmark Library.

2.1.1.1 zChaff vs BerkMin561

We ran zChaff (2001.2.17 version) and BerkMin561 on the IBM CNF Benchmark. We used the default configuration for both engines. For each model, we used SAT dat.k.cnf with k=1, 10, 15, 20, 25, 30, 35, 40, 45, 50. The time-out was set at 10 000 seconds on a workstation with 867841X Intel(R) Xeon(TM) CPU 2.40GHz. The following is a summary of the results obtained:

	zChaff	BerkMin561
Total time (10000 sec timeouts)	344765	414094
First (# of CNF where the engine is the fastest)	298	131
Timeout number	25	30
+ (# of CNF where the engine is the fastest by more than a minute and 20%)	67	32
First by model (# of models where the engine is the fastest)	28	18

Table 2.1. The results are displayed in seconds. The timeout was set to 10000 seconds.

More complete experimental results are presented in Table 2.2, where for each model in the table, we present the sum of the results of SAT dat.k.cnf with k=1, 10, 15, 20, 25, 30, 35, 40, 45, 50 (ie the BMC translation of each model for the different k's). For more details, see the complete results in [ICBI]. Because SAT solvers do not always behave in an homogeneous manner (Cf. detailed results in [ICBI] or table 2.3), the complete results analysis should not be disregarded.

The results show that zChaff and BerkMin561 achieved close results. On some CNFs, zChaff runs faster, and on others, BerkMin561 runs faster. In most cases, the differences in their performances is not significant (i. e., the faster speed is not faster by more than one minute or 20%). However, while zChaff seems to perform slightly better overall, BerkMin561 gets better results than zChaff on the UNSAT CNFs.

From these results, it is not possible to conclude that BerkMin561 performs better than zChaff. This is not consistent with what can be read in the literature or with the final results of the SAT03 contest (Cf. 2.3).

2.1.1.2 Siege

Siege was hors-concours for the SAT03 contest. Therefore it did not participate in the second stage. Nevertheless, the siege results were pretty good for the first stage and siege has the reputation for being one of the best SAT solvers available. We ran siege_v4 on the benchmark using 123456789 as a seed.

Table 2.4 displays the overall conclusions. For more details see Table 2.2. Siege_v4 is the fastest in 298 cases (CNFs) out of 498. In many difficult cases, siege_v4 is the fastest by an order of magnitude or more. In some cases, siege_v4 performed significantly worse than zChaff or BerkMin561 in a significant way (e. g., for 26 rule, siege_v4 is slower than zChaff by an order of magnitude and slower than BerkMin by two orders of magnitude).

In conclusion, we see that siege_v4 performs significantly better than zChaff on the IBM CNF benchmark. In addition, within the timeout, several CNFs can be solved only by siege_v4. However, siege is a randomized solver hence it can be argued that the comparison is not fair since the zChaff and BerkMin561 versions we used were not randomized (though the versions of these two solvers that participated in the SAT03 contest were not randomized). Nevertheless, even a deterministic solver can be lucky and running siege with a fix random seed makes it deterministic.

2.1.1.3 Comparison with SAT03 contest results

In order to try to understand whether or not our results are consistent with those of the SAT03 contest, we had a look at the results of the first and second stages of the competition.

First stage

When we look at the results 4 for the industrial category, we note the following:

• Series 13_rule_ is over represented. Besides, since all solvers timeout on most of the CNFs from this series, it is not very meaningful.

• Except for series 13_rule_ and 11_rule_, the series from IBM CNF benchmark are easy for zChaff, BerkMin561 and siege_v1.

• Results for rule_07 are not consistent with our own experiments. Even if Table 2.5 is somehow biased by the fact that we did not use the same hardware as the SAT03 contest for our experiments, it clearly points out that there is a strong discrepancy for series 07_rule_. This discrepancy is probably caused by the "Lisa syndrome": CNFs were shuffled for SAT03 and solvers performance can change dramatically between a shuffled CNF and the original. We believe that for SAT03 07 rule series, zChaff was a "victim" of the Lisa syndrome but BerkMin561, forklift and siege were not affected, or at least not in such an order of magnitude. Indeed, the results we get on 07 rule with BerkMin561 and siege_v4 are consistent with the SAT03 results.

The SAT03 contest results for BerkMin561, forklift, siege_v1, and zChaff on (shuffled) series from IBM CNF Benchmark are summarized in Table 2.6. If results from series 07 rule and 13 rules are discarded, zChaff gets better results than BerkMin561. We believe that the result differences between the first stage of SAT03 contest results for IBM benchmark and our experiments are due to clauses shuffling in SAT03 and to the fact that the SAT03 experiment used a smaller test bed of CNFs from IBM benchmark.

Second stage

Table 2.7 gives some results from the second stage of the SAT03 contest on industrial benchmarks. Clearly, the respective zChaff and BerkMin561 ranking do not correspond with our experimental results (see Table 2.1). However, in the second stage, all solvers "timed-out" on the IBM benchmarks. In other words, the ranking of the solvers selected for the second stage of the competition did not take into account performances on IBM benchmarks. This probably explains why our evaluation of zChaff and BerkMin561 on the IBM CNF Benchmark gives results that are not in line with the second stage results (on industrial benchmarks) of the SAT03 contest.

2.1.2 Lessons for benchmarking

Our experimental results, especially the comparison between zChaff and BerkMin561, are not in line with other results, such as those from SAT03. In this section we try to understand why and attempt to learn from this.

Use relevant benchmarks

To compare two SAT solvers for BMC, it is extremely important to use relevant benchmarks from BMC applications. For instance, results from theoretical problems such as 3-SAT problems, hand-made problems, or problems from applications others than BMC (such as planning), are not relevant and will not provide a good comparison between SAT solvers performances for BMC applications.

It is not sufficient to use only the CNFs generated by BMC; the models from which the CNFs are generated should be relevant. Therefore, it is of prime importance to use models from real-life industrial verification projects.

Use relevant timeout

The timeout used for the SAT03 contest was 600 seconds or 900 seconds for the first stage, and up to 2400 seconds for the second stage of the competition. For our experimentation, we used a 10000 seconds timeout. From the detailed experimental results it is clear that the lower the timeout, the less relevant the results. Additionally, the lower the timeout, the more timeout results would be received for all the SAT solvers. Consequently, these timeouts can actually cover up different situations. For example, in our experiments for CNF k45 from model 11 rule 1, with a timeout of 900 seconds, zChaff and BerkMin561 would both get time out. However with a 10000 second timeout, we realized that one solver performs seven times faster than the other on this CNF.

The ideal solution would be not to use any timeout or at least use very long timeouts (e. g. one week). In this way, the results for the SAT solvers can always (or almost always) be compared for a given CNF (at least a "reasonable" CNF). The obvious drawback is that this makes the experiments very, very long, hence limiting the scope of such experiments. In the SAT contests, the choice of a short timeout allows a great number of SAT solvers to be run against a very broad spectrum of CNFs. In our experiments, we ran a very limited number of SAT solvers against a smaller (but more relevant for BMC) number of CNFs with a longer timeout. We believe that this is one of the main explanations for the difference in results between SAT03 and our results. More precisely, if a longer timeout would have been used for the second stage of the SAT03 contest, probably some solvers would have solved of the benchmark from the IBM CNF Benchmark; therefore, the final results would more likely have been consistent with our experiments. In summary, using shorter timeouts can be very useful for larger scopes experiments (many SAT solvers and very broad benchmark). However, in order to get more relevant results for BMC7, these kinds of experiments must be refined by limiting the scope (e. g., the number of SAT solvers).

Shuffling clauses in benchmark CNFs is tricky

Shuffling clauses in benchmark CNFs can have a dramatic effect and potentially changes solvers performance ranking. However, in real life, SAT users don't shuffle their CNFs. One of the major differences between our experiments and the SAT03 experimentation is that we did not shuffle CNFs.

Use a broad benchmark: easy for a SAT solver, does not means easy for all

When assessing the performance of a new SAT solver, a common trend is to run the new solver against CNFs that are difficult to solve with a well established solver (e. g. zChaff). Although it

seems reasonable, this can be very misleading. Modern SAT solvers rely heavily on heuristics; therefore a CNF can be solved very easily with one solver and still be very difficult with others. For example, the CNFs from model 01 rule are very easy for zChaff to solve but difficult for BerkMin561. On the 01 rule series, zChaff typically runs faster than BerkMin561 by two orders of magnitude. On the 26 rule series siege_v4 typically runs worse than zChaff by one order of magnitude and worse than BerkMin561 by two orders of magnitude. The reason for this kind of behavior is not that some models are “special”, but more likely the limitation of the heuristics used by different solvers. The fact that solvers performance ranking often varies for different CNFs generated from the same model comforts us with the idea that the relationship between solvers (and heuristics) performances and model specifics is not as strong as one would think.

For results analysis, there is no real reason to discriminate between results for satisfiable and unsatisfiable CNFs

We believe that it does not make much sense to differentiate between the performance of solvers for satisfiable and unsatisfiable CNFs – at least for BMC applications. Firstly, BMC is usually run in an incremental manner (e. g., k = 0 . . . 10, k = 11 . . . 15, . . .). Therefore, before you can get a satisfiable result you often have to get several unsatisfiable results. Secondly, some models cannot be falsified and CNFs generated by BMC will always be unsatisfiable. Therefore, only the global performances are really relevant for real-life BMC SAT use.

Use several metrics for comparisons

Since SAT solvers rely heavily on heuristics, it is unlikely (or at least rare) that a SAT solver would be better on all possible CNFs (from real-life models). For example, even though siege_v4 performs better than zChaff and BerkMin561 in most CNFs of the IBM benchmark, it does not perform better for all (e. g. model 26 in Table 2.2). The following metrics can be used to compare two SAT solvers, solver 1 and solver 2:

• Global time. This presents two drawbacks: 1) There is no perfect solution to take timeouts into account, and 2) All the weight is on the CNFs which take the longest to be solved.

• Ratio between the number of CNFs solved more quickly by solver 1 and the number of CNFs solved more quickly by solver 2. Optionally, only the cases where performance differences are significant can be taken into account.

• The average of the ratio between solver 1’s performance and solver 2’s performance on each CNF. This presents two drawbacks: 1) There is no perfect solution to take the timeout into account, and 2) All the CNFs have the same weight.

• Timeout numbers. It is quite relevant if one considers that the timeout value is roughly equivalent to the real-life timeout (i.e. the maximum reasonable waiting time of the real-life formal tool users).

2.1.3 Conclusion

Benchmarking is not a trivial task and it can be misleading. Our experiments on zChaff and BerkMin561 present results that are contradictory with what is commonly accepted by the SAT community (i.e. that BerkMin561 would outperform zChaff).

It is diffcult to compare two SAT solvers (e. g. CNFs that are difficult for zChaff are not the same as CNFs that are diffcult for BerkMin561). Even when a SAT solver such as siege_v4 seems to clearly outperform BerkMin561 and zChaff, it is not necessarily so for all benchmark CNFs. In order to help benchmarking and compare between solvers results, we proposed guidelines that sketch a methodology. We believe that the systematic use of this benchmarking methodology can improve the general quality of experimental performance evaluations for SAT BMC and will help to produce practical and fundamental advances in the area.

In near future we plan to use our methodology to benchmark the winners of the SAT04 contest and the updates of “old” SAT solver (e. g. SAT04 version of zChaff).

Though we tackled only the SAT part of BMC formal verification, the actual BMC (translation from a bounded model to a CNF) is very important. First, BMC translation runtime can sometimes make an actual difference. Second, and most importantly, solvers performance actually depends heavily on the quality of the CNFs generated.

Model	**zChaff**	**BerkMin561**	**siege_v4**	*Model*	**zChaff**	**BerkMin561**	**siege_v4**
01_	166	22800(1)	262	***14__2***	3490	2480	378
02_1__1	347	34	38	***15_***	6	21100(1)	1
02_1__2	336	69	104	***16_2__1***	0	10	0
02_1__3	26	16	10	***16_2__2***	0	9	1
02_1__4	34	12	13	***16_2__3***	0	11	0
02_1__5	32	16	19	***16_2__4***	3	16	0
02_2_	118	83	11	***16_2__5***	1	17	2
02_3__1	96	157	22	***16_2__6***	1	18	1
02_3__2	702	34	13	***17_1__1***	0	509	0
02_3__3	154	133	18	***17_1__2***	219	427	70
02_3__4	333	39	13	***17_2__1***	1	28	0
02_3__5	126	172	25	***17_2__2***	1	98	0
02_3__6	396	34	16	***18_***	32200(2)	31200(2)	5160
02_3__7	262	86	39	***19_***	127	2910	482
03_	190	733	126	***20_***	21800(1)	9590	4020
04_	170	597	161	***21_***	150	2245	383
05_	33	267	124	***22_***	5290	5980	582
06_	296	1140	130	***23_***	38700(2)	28500(1)	2820
07_	61	73	69	***26_***	155	26	2850
09_	7	1	2	***27_***	53	364	134
11__1	5080	21900(1)	1760	***28_***	385	843	96
11__2	6640	14500(1)	982	***29_***	35700(2)	58200(5)	16700
11__3	24100(2)	24000(2)	3135	***30_***	76600(7)	72000(7)	66000(5)
12_	0	0	0				
13__1	90000(9)	90000(9)	90000(9)				
14__1	191	719	126				

Table 2.2 The results are displayed in seconds. The timeout was set to 10000 seconds. For each model, the number of timeouts, if any, appears in brackets.

18_ruleCNF	***Result***	***zChaff***	***BerkMin***	***Siege***
SAT_dat.k1.cnf	unsat	0	0	0
SAT_dat.k10.cnf	unsat	1	1	0
SAT_dat.k15.cnf	unsat	13	4	5
SAT_dat.k20.cnf	unsat	65	47	41

SAT_dat.k25.cnf	unsat	951	109	251
SAT_dat.k30.cnf	sat	700	755	557
SAT_dat.k35.cnf	sat	timeout	2230	1730
SAT_dat.k40.cnf	sat	5510	8060	247
SAT_dat.k45.cnf	sat	timeout	timeout	959
SAT_dat.k50.cnf	sat	5010	timeout	1370

Table 2.3 The results are displayed in seconds for the CNFs from 18 rule model. The timeout was set to 10000 seconds. zChaff performs the worst for k=15, 20, 35; BerkMin561 performs the worst for k=30, 40, 50; Siege 4 has the best performance except for k=15, 25.

	total time (including 10000sec timeout)	*# timeout*
zChaff	345000 sec	25
BerkMin561	414000 sec	30
Siege_v4	187000 sec	14

Table 2.4 The timeout was set to 10000 seconds.

	SAT03	***SAT03***
SAT_dat.k1.cnf	0	0
SAT_dat.k10.cnf	>900	11
SAT_dat.k15.cnf	>900	5
SAT_dat.k30.cnf	>900	6

Table 2.5 zChaff CPU runtime in sec for SAT03 contest 07 rule series.

	BerkMin561	**forklift**	**Siege_1**	**zChaff**
Total # of Solved Benchmarks	112	112	112	101
Total CPU time needed (sec)	105000	103000	10300	114000
without 13 rule (sec)	1510	518	408	4160
without 13 rule and 07 rule (sec)	1410	424	268	553

Table 2.6 Partial results from first stage SAT03 contest

Rank	*Solver*	*#Solved*
1	Forklift	12
2	BerkMin561	11
6	zChaff	4

Table 2.7 Partial second stage SAT03 contest results on industrial benchmarks

2.2 *Comparison between ESolver, Siege and DPSAT (Infineon)*

2.2.1 Test set-up

We present the results for our experimental SAT solver (E-Solver), one of the best publicly available solver (Siege V4), and our currently productive solver DPSat. The set of benchmarks used is the IBM-BMC benchmark consisting of 45 groups of CNF problems, with each group containing in the order of 20 instances. The tests were run on an Intel(R) Xeon(TM) CPU 2.66GHz, with 512 KB first level cache, and 2 GB physical memory under Linux. More than 200 CPU hours of computer time was used for the tests. For most instances, a resource limit of 10000 sec was used. If the algorithm did not terminate within that, the process was killed.

2.2.2 Results

2.2.2.1 Raw data

A total of 936 test cases were run, most of them with each of the three tools. The results were collected in a large table, of which we include two random sections below. Each section collects all instances of one problem class. The columns labelled E-Solver, Siege_V4 and DPSat give the computation times of the respective tool in seconds. The next column says whether the instances is satisfiable or unsatisfiable.

The last two columns compare E-Solver to the best of the other two on each instance: the "Best" column repeats the lower of the two figures for Siege and DPSat, and "ESolver/Best" gives the quotient of the computation times.

Instance	E-Solver	Siege_V4	DPSat	S/U	Best	Esolver/Best
01_rule:SAT_dat.k10:	0,24	0,04	0,85	u	0,04	6,00
01_rule:SAT_dat.k15:	0,87	0,33	1,25	s	0,33	2,64
01_rule:SAT_dat.k1:	0,00	0,01	0,05	u	0,01	0,00
01_rule:SAT_dat.k20:	4,92	2,20	4,50	s	2,20	2,24
01_rule:SAT_dat.k25:	14,31	1,41	5,65	s	1,41	10,15
01_rule:SAT_dat.k30:	20,72	111,72	34,15	s	34,15	0,61
01_rule:SAT_dat.k35:	34,08	59,05	87,03	s	59,05	0,58
01_rule:SAT_dat.k40:	68,82	19,08	211,23	s	19,08	3,61
01_rule:SAT_dat.k45:	18,77	82,61	108,52	s	82,61	0,23
01_rule:SAT_dat.k50:	74,29	120,71	277,99	s	120,71	0,62
01_rule:SAT_dat.k55:	121,05	173,05	365,78	s	173,05	0,70
01_rule:SAT_dat.k60:	172,02	670,99	419,12	s	419,12	0,41
01_rule:SAT_dat.k65:	405,05	267,53	465,86	s	267,53	1,51
01_rule:SAT_dat.k70:	266,59	126,91	2167,20	s	126,91	2,10
01_rule:SAT_dat.k75:	357,19	343,09	1178,03	s	343,09	1,04
01_rule:SAT_dat.k80:	814,58	678,73	1795,83	s	678,73	1,20
01_rule:SAT_dat.k85:	1038,83	476,54	4246,15	s	476,54	2,18
01_rule:SAT_dat.k90:	252,81	547,49	1143,03	s	547,49	0,46
01_rule:SAT_dat.k95:	842,81	1884,67	4674,76	s	1884,67	0,45
01_rule:SAT_dat.k100:	4590,83	2474,72	1221,02	s	1221,02	3,76
total	**9098,78**	**8040,88**	**18408,00**		**8040,88**	**1,13**

Instance	E-Solver	Siege_V4	DPSat	S/U	Best	Esolver/Best
05_rule:SAT_dat.k10:	0,12	1,00	1,35	u	1,00	0,12
05_rule:SAT_dat.k15:	0,43	1,00	2,50	u	1,00	0,43
05_rule:SAT_dat.k1:	0,00	1,00	0,02	u	0,02	0,00
05_rule:SAT_dat.k20:	0,19	1,00	3,79	u	1,00	0,19
05_rule:SAT_dat.k25:	0,49	1,00	6,07	u	1,00	0,49
05_rule:SAT_dat.k30:	0,39	3,00	8,54	u	3,00	0,13
05_rule:SAT_dat.k35:	1,65	8,00	17,63	s	8,00	0,21
05_rule:SAT_dat.k40:	2,28	22,00	18,09	s	18,09	0,13
05_rule:SAT_dat.k45:	4,38	37,00	29,33	s	29,33	0,15
05_rule:SAT_dat.k50:	4,83	60,00	37,37	s	37,37	0,13
05_rule:SAT_dat.k55:	5,48	142,00	43,34	s	43,34	0,13
05_rule:SAT_dat.k60:	8,30	151,00	63,25	s	63,25	0,13
05_rule:SAT_dat.k65:	13,15	219,00	86,53	s	86,53	0,15
05_rule:SAT_dat.k70:	11,13	249,00	105,18	s	105,18	0,11
05_rule:SAT_dat.k75:	21,09	506,00	126,21	s	126,21	0,17
05_rule:SAT_dat.k80:	30,23	559,00	244,52	s	244,52	0,12
05_rule:SAT_dat.k85:	16,26	538,00	124,41	s	124,41	0,13
05_rule:SAT_dat.k90:	38,70	1162,00	273,90	s	273,90	0,14
05_rule:SAT_dat.k95:	55,12	852,00	183,56	s	183,56	0,30
05_rule:SAT_dat.k100:	79,98	1110,00	541,40	s	541,40	0,15
total	**294,20**	**5623,00**	**1916,99**		**1916,99**	**0,15**

Instead of copying the full result tables here, we give a more condensed view in the next paragraph.

2.2.2.2 Statistics

Unsolved instances:

11 (of 936) instances were not solved within reasonable resource bounds (10000 sec) by neither DPSat nor Siege. All but two of these were solved by E-Solver.

Solved instances

The rest of the analysis takes only those instances into account which were solved by E-Solver and at least one other tool. For those we counted the number of instances for which E-Solver was better by certain factors compared to the better of the other two. In the following table, the column "<K" means that in 729 instances out of 924, i.e. 78,9%, E-Solver was faster than K times the run-time of the better of the other tools. It is remarkable that in many instances (34% resp. 6 %) it is much faster, i.e. by a factor of 5 resp. 100.

As it is well known that such results show a wide variance, we included the "<2" column: it can be taken to say that in 78.9% the new approach is at least reasonably good compared to the other two.

# solved instances	<2	<1	<0,2	<0,01
924	729	630	321	56
100,0%	78,9%	68,2%	34,7%	6,1%

Hard instances

It can be argued that a performances gain even by a factor of 100 is not really too relevant if the required time is in the seconds range anyway. We therefore defined “hard” instances to be those that required at least 10 sec by the previous tools, and did the same evaluation on this population, with the following result:

#hard instances	<2	<1	<0,2	<0,01
405	387	351	191	14
100,0%	95,6%	86,7%	47,2%	3,5%

Very hard instances

If, in the same line of reasoning, we take “very hard” to mean more than 1000sec, the picture remains quite consistent with the above:

#instances	<2	<1	<0,2	<0,01
78	74	64	24	10
100,0%	94,9%	82,1%	30,8%	12,8%

Satisfiable vs unsatisfiable.

Similar evaluations were done for the satisfiable and unsatisfiable instances, but no significant difference arose, i.e. the performance gain is approximately equal regardless of satisfiability of the instance.

Average performance gain

On the set of “hard” instances, we computed the average of the performance ratios (E-Solver/Best). It is 0.37, i.e. the average performance gain can be said to be a factor of (1/3.7=) **2.7**.

2.3 Experimental results and conclusions

2.3.1 Comparison technique

We made a comparison among five SAT solvers: zChaff, BerkMin561, zChaff II, Siege V4 and E-Solver. Due to technical reasons we did not run all the SAT solvers on the same machine.

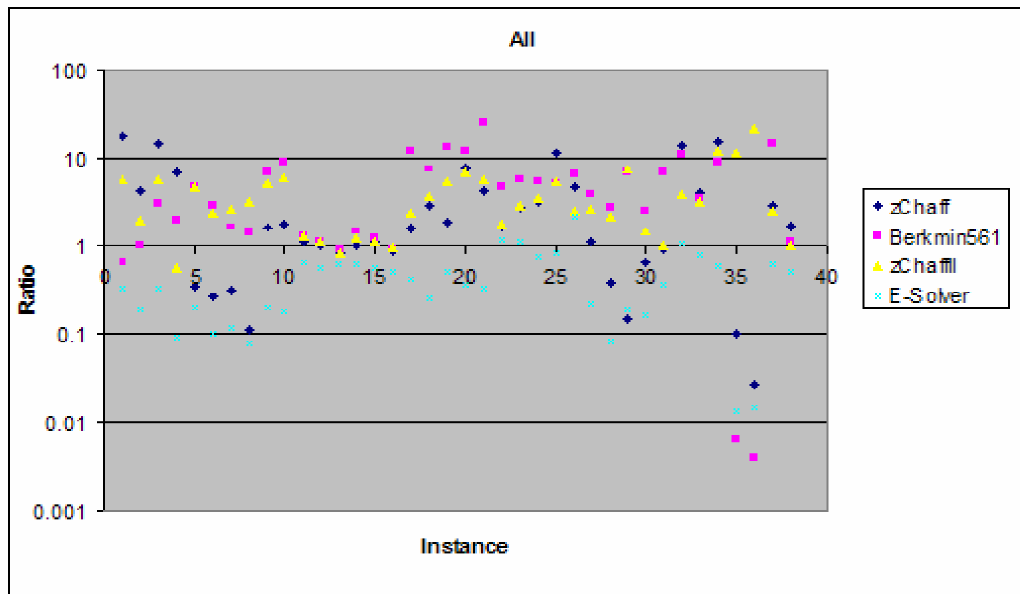
- Infineon ran E-Solver and Siege V4 on an Intel(R) Xeon(TM) CPU 2.66GHz, with 512 KB first level cache, and 2GB physical memory on Linux.
- IBM ran zChaff, BerkMin561, zChaff II and Siege V4 on an Intel Xeon dual CPU 2.4GHz, with 512 KB first level cache, and 2.5GB physical memory, also on Linux.

In order to compare the SAT solvers we used Siege V4 runtime as a common denominator for our experiments and based our comparison on the relative speed of each SAT solver to Siege v4. Hence we normalize, for each CNF instance, the runtime of each solver by the runtime execution of Siege V4 on the same machine. After normalization the ratio of Siege V4 is 1, the ratio of

SAT solver that take longer than Siege V4 on the same rule are > 1 and the ratio of runtime executions that take shorter time than Siege V4 are < 1. The runtime ratio between IBM and Infineon machines for each execution of Siege V4 lay in the range 0.9 and 1.1.

2.3.2 Join Results

We picked 39 instances from IBM benchmark on which we base our comparison. They all fulfil the ratio requirement of having the Siege v4 runtime executions on the two machines in the range 0.9 – 1.1. The following chart



represents the solvers runtime executions normalized with Siege V4.

- ***zChaff***, stands for the first zChaff version that was released by Princeton.
- ***zChaff II*** stands for the new zChaff, a version that was released at 13.5.2004.

We used a logarithmic scale to represent the runtime ratio. Due to the comparison process, in which we use Siege v4 as the base of our comparison, Siege V4 ratio is 1 for all instances, hence we removed it from the chart. According to our experiments E-Solver has the best performance overall. In 35 of the instances E-Solver runtime is shorter than Siege V4 and in most cases the speedup ratio is up to one order of magnitude. In 3 of the instances, Siege V4 runtime is better than E-Solver, but the speed up ratio in these cases is no more than 2.2 for Siege V4. We conclude that on our benchmark E-Solver performance is consistently much better than the other SAT solvers. Siege V4 is the best SAT solver only for 3 instances. According to our results Siege V4 takes the second place of the best SAT solver, among the SAT solvers used in our experiments.

zChaff performs better than Siege V4 in 10 of the CNF instances. Unlike E-Solver, zChaff results are not consistent throughout the benchmark and vary from slowdowns of ~18 to a speedup of almost two orders of magnitude. The results for zChaff remain not consistent even when we compare it to the rest of the SAT solvers. To be noticed that zChaff has the worst performance in 9 of the instances. We must also note that even in cases where zChaff outperforms Siege V4, E-Solver is usually (excluding one instance) better than zChaff on these instances.

BerkMin561 performs better than Siege V4 in 4 of the CNF instances. In most of the examples BerkMin561 performs much worse than Siege V4, getting even slowdowns of ~25. In two examples BerkMin561 performs better than Siege V4 by more than two orders of magnitude. BerkMin561 performance is also not consistent, the result depending a lot on the cnf run. To be noticed that in 22 CNF instances BerkMin561 is the slowest among the participating SAT solvers.

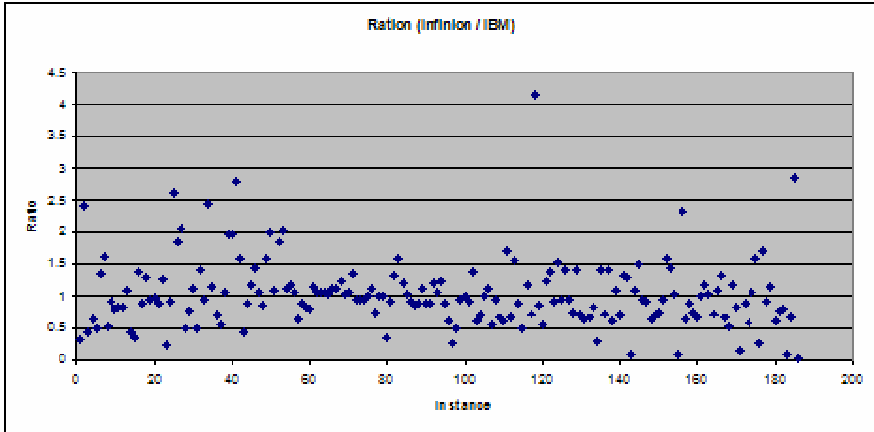
zChaff II performs better than Siege V4 in only 4 of the CNF instances, and on those occasions its speedup over Siege V4 is never higher than 2. On the other hand zChaff II appears to be the worst among all the participating SAT solvers only in 4 instances, which is much better than the other participating (excluding E-Solver). Overall, zChaff II looks better than BerkMin561.

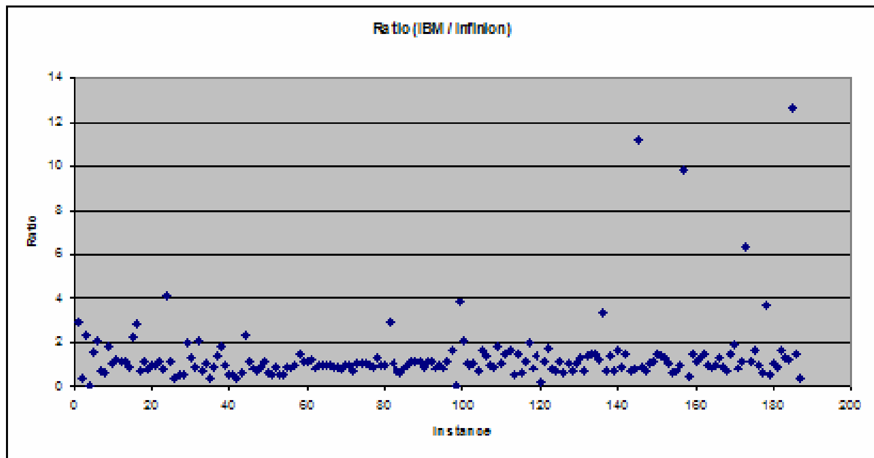
After analyzing the results, we conclude that E-Solver has no competitors among the participating SAT solvers. Only in 6 instances E-Solver does not provide the best result, but even in these cases only one SAT solver performance was better than E-Solver. Siege V4 clearly takes the second place as the best SAT solver, it is clear to see that Siege V4 performance is better than the rest of the solver (excluding E-solver) in most of the instances). There is no clear conclusion regarding the rest of the SAT solver, because their results are similar and the benchmark is not large enough to be able to learn a definite relation among their performance.

2.3.3 Heuristics and randomness effects

Siege is a SAT solver that can run in a random mode; its heuristics are affected by a random value. If provided a seed, Siege runs are deterministic, if not, it behaves randomly. As we noticed before, one of the problems in SAT solvers is an inconsistency in their result. Depending on the cnf, SAT solvers can show varying performances. Heuristics success is generally highly dependent on the cnf they run on. Looking at the same problem from a different angle, we can state that the same cnf can be solved faster or slower by the same SAT solver with a slight change in the heuristics it uses.

The effect of randomness is not a new issue. What is surprising is how strong this influence is, as seen in the next charts. On one hand we have Siege V4 run deterministically with a fix seed on all the benchmark. On the other hand we have the results of it being run randomly, each time with a different seed chosen by the system. Our experiments show difference in performance ranging from a speed up of x52 to a slowdown of x12.65. It proves that using randomness while searching for the satisfying assignment can be in the same time power and problematic.


Ration (Infinion / IBM)
Ratio
Instance
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
20
40
60
80
100
120
140
160
180
200


Ratio (IBM / Infinion)
Ratio
Instance
14
12
10
8
6
4
2
0
0
20
40
60
80
100
120
140
160
180
200

3 The Improvements of SAT based Bounded Model Checking

Proper benchmarking, which matches day to day use of formal methods, allows us to assess direct improvements for SAT use for formal methods. Proper use of our benchmark allowed us to prove that previous results on tuning SAT solver for Bounded Model Checking (BMC) were overly optimistic and that a simpler algorithm was in fact more efficient.

Over the past decade, verification via model checking has evolved from a theoretical concept to a production-level technique. It is being actively used in chip design projects across the industry, where formal verification engineers can now tackle the verification of large industrial hardware designs. Assessing theoretical results requires more than simply getting experimental results on some benchmarks. Rather, results must be attained for a wide range of benchmarks, under conditions as close as possible to real-life formal verification.

In order to illustrate the significance of proper benchmarking, we demonstrate how applying overly restrictive benchmarking can be misleading. We show how proper benchmarking, which matches day-to-day use of formal methods, allows us to assess direct improvements. We present a method of splitting up the problem, which verification engineers usually meet in day-to-day real-life work. This allows us to distribute the problem in a simple and efficient way. This distribution technique is simple, but we, nevertheless, prove that it is very efficient by benchmarking. In a more general way, our goal is to show how very simple ideas can be proven to have great impact by experimentation.

3.1 Decision Heuristics for Tuning SAT

This section presents some decision heuristics for tuning the zChaff VSIDS for bounded model checking as in [**SZ**]. The experimental results use a predecessor of the IBM Formal Verification Benchmark. The next section shows how proper use of the IBM Formal Verification (Cf. appendix A). Benchmarking proves the results in this section are too optimistic.

3.1.1 Bounded Model Checking and SAT Basis

BMC basis: BMC translates a safety formula from LTL [**Pn**] into a propositional formula under bounded semantics. The general structure of a $G(P)$ formula, as generated in BMC [**Bi2**], is as follows:

$$\phi: \quad I_0 \wedge \bigwedge_{i=0}^{k-1} \rho(i, i+1) \wedge (\bigvee_{i=0}^{k} \neg P_i)$$

where I_0 is the initial state, is $\rho(i, i+1)$ the transition between cycles $i$and i+1, and $P_i$is the property in cycle i. If this propositional formula is proven to be satisfiable, the satisfying assignment provided by the SAT solver is a counterexample to the property $G(P)$. To convert the initial propositional formula into Conjunctive Normal Form (used as the input format by most SAT solvers), extra variables are introduced to avoid combinatory explosion. Usually, these extra variables represent more than 80% of the total number of variables in the CNF formula.

SAT basis: SAT is the problem of determining the satisfiability of a Boolean formula. The problem was used by Cook to define NP-completeness [**Co**]. Today, many implementations are available for solving the problem, such as Grasp[SS] and zChaff [**Mo**]. Most of them are based on the complete DPLL algorithm [**DLL**], we now

describe as in [**Mo**]:

```
while(true) {
    if (!decide()) // if no unassigned vars
        return(satisfiable) ;
    while (!bcp()) {
        if (!resolveConflict ())
            return (not_satisfiable) ;
} }
bool resolveConflict() {
    d=most recent decision not 'tried both ways' ;
    if (d==null) // no such d was found
        return false ;
    flip the value of d;
    mark d as tried both ways ;
    undo any invalidated implications;
    return true;
}
```

decide() is a function that chooses the next variable according to which branching will occur. There are many heuristics for choosing this next variable, such as DLIS (Dynamic Largest Individual Sum) and VSIDS (Variable State Independent Decaying Sum). zChaff uses VSIDS as its decision heuristic. *bcp*() returns true when the boolean constraint propagation (*bcp*) [**Mo**] finishes without conflict. Our next subsection focuses on tuning this heuristic for BMC.

3.1.2 Tuning VSIDS

The original zChaff decision VSIDS strategy is as follows:

1. Each variable in each polarity has a counter, which is initialized to 0.

2. When a clause is added (by learning) to the database, the counter associated with each literal in the clause is incremented.

3. The (unassigned) variables and polarity with the highest counter are chosen at each decision.

4. Periodically, all counters are divided by a constant.

One of Strichman's [**St**] main idea is as follows: in the Davis-Putnam decision procedure, the variable of the original propositional formula is used first and in a specific static order. This static order is determined by a breadth first search of the (k-unfolding of the) variable dependency graph; the search starts from the set $\bigcup_{0\leq i\leq k} \neg P_i$. Roughly speaking the intuition behind it is that the formula variables are the most critical. We chose to implement several decision heuristics on top of zChaff [1]. We tuned the zChaff VSIDS decision heuristic in several different ways. First, we wanted to reflect the idea of static order (SO) [**St**]. Second, we wanted to implement a heuristic that gives priority to dominant variables (DV) over any other variables, but otherwise relies on zChaff decisions. This is because it is unclear from [**Si**] whether the improvements were due to the static order or to the priority given to the dominant variables (*i.e.*,

the variables from the initial propositional formula before the conversion to CNF, or domain variables). Because zChaff uses the VSIDS decision strategy, which is less time consuming than the decision heuristic used by Strichman with GRASP (*i.e.*, DLIS), it is not clear whether the benefits from a static order would still be realized. Therefore, we also implemented a static order heuristic and dominant variables priority heuristics to act as a tie breaker for the zChaff decision (respectively SB and DVB). The four decision strategies implemented are described in details in [**SZ**].

3.1.3 Experimental Results

In our first experiments, we used a predecessor of the IBM Formal Verification Benchmark. For each model we used only one CNF (see Table 3.1 contend for experimental results). In Table 3.2, we computed speedup, taking the VSIDS heuristic as a reference: therefore we did not take into account IBM_16 and IBM_19, for which VSIDS times out. We could also have decided to take the Static Order (SO) heuristic as a reference and not take IBM_10 and IBM_17 into account. However, SO is the only heuristic that times out for IBM_10 and IBM_17, while three out of five heuristics time out for IBM_16 and IBM_19. When we compared min(DVB,SB,DV,SO) with VSIDS for each case, we were impressed: Running four concurrent instances of SAT, each with a different heuristic, should give a theoretical speedup greater than six. Indeed if we could build a tool which would run concurrently SAT instances with DVB, SB, DV and SO heuristics, this would solve SAT problems about six times faster than with VSIDS heuristics[2].

	VSIDS	DVB	SB	DV	SO	min(DVB,SB,DV,SO)
1_2001	18	12	31	14	170	(DVB)12
2_2001	1400	620	820	300	1300	(DV)300
3_2001	43	66	190	240	240	(DVB)66
4_2001	4000	1000	800	210	120	(SO)120
5_2001	2400	44	240	170	100	(DVB)44
6_2001	1000	67	190	3800	250	(DVB)67
7_2001	340	120	140	8	19	(DV)8
8_2001	13	10	34	14	4800	(DVB)10
9_2001	71	44	110	190	900	(DVB)44
10_2001	70	70	110	86	timeout	(DVB)70
11_2001	4800	4200	6400	timeout	3100	(SO)3100
12_2001	44	42	73	46	51	(DVB)42
13_2001	78	6	58	6	52	(DVB)6
14_2001	32	31	42	18	26	(DV)18
15_2001	13	13	13	13	13	13
16_2001	timeout	timeout	9200	timeout	180	(SO)180
17_2001	7600	4000	3100	2400	timeout	(DV)2400
18_2001	11	85	95	6	3000	(DV)6
19_2001	timeout	timeout	timeout	1600	8700	(DV)1600

Table 3.1. The results are displayed in seconds, with two significant digits. Timeout was set to 10000 seconds.

	VSIDS	DVB	SB	DV	SO
Total time	21933	10430	12446	>17521	>34141
Global speedup (VSIDS total time/total time)		2.10	1.76	<1.25	<0.64
Average speedup (average VSIDS time/time)		6.11	2	<6.26	<5.15

Table 3.2. The results are computed without IBM_16 and IBM_19 for which VSIDS times out

In order to evaluate this claim, we made some additional experiments, with a different approach. First, we decided to use the IBM Formal Verification Benchmark, which is wider than its predecessors; the fact that it is available online (for academic organizations) allows the reproduction of our results. Second, we decided not to run SAT as a stand-alone with one or two CNFs per model, but to run our global bounded model checking tool, as a regular user would, without any prior knowledge of the model. We began with a bound equal to 10, and in the case where no satisfiable assignment is found for the CNF generated by BMC translation, we incremented the bound by 5, and so on. In other words, in order to try to falsify a safety formula, say $G(P)$), we usually try to find a satisfiable assignment for

$$\phi_0 : \quad I_0 \wedge \bigwedge_{i=0}^{9} \rho(i, i+1) \wedge (\bigvee_{i=0}^{10} \neg P_i)$$

if it fails, we try

$$\phi_1 : \quad I_0 \wedge \bigwedge_{i=0}^{14} \rho(i, i+1) \wedge (\bigvee_{i=11}^{15} \neg P_i)$$

and keep incrementing until a satisfiable assignment is found for some ff_j, some timeout is reached, or a user defined maximal bound is reached. This is the most common approach for day-to-day use of BMC. Even when the completeness threshold d(as in [**Bi2**]) is known, it is very often too big to allow the BMC to finish with k= d. Therefore, most of the time, users will try to falsify a formula with BMC, without knowing whether the formula holds or not, or which $k$would be needed to get a satisfiable ff_k.

We ran RuleBase with five concurrent SAT BMC instances (each of them using a different decision heuristic from VSIDS, DV, DVB, SB, SO). Each instance was independent and run on a single workstation with a 867841X Intel(R) Xeon(TM) CPU 2.40GHz with Red Hat Linux release 7.3. Surprisingly, we got a speedup of only 1.14 (see the following section and Concur 5/5 heuristics results in Table 3.4). This can be explained by several factors:

The search includes a SAT search together with pre-processing and BMC translation (translation from the model with a given bound to a CNF). SAT is only a part of the whole process, however the bottom line for formal verification is performance of this global process.

For each model, several CNFs (from BMC translation with different bounds) are searched for satisfiability. All in all, SAT was run in many more CNFs than for the 2001 benchmark (several CNFs from the 2001 benchmarks are generated from the IBM Benchmark). Running SAT on a benchmark that was too small did not accurately reflect

the conditions of day-to-day formal verification. This led us to anticipate overly optimist conclusions.

3.2 A Simple but Efficient distributed SAT BMC Algorithm

We conducted several experiments with the IBM Formal Verification Benchmark. In fact, it has become one of our main tools to assess new search engines and algorithms. The experiments we present here were conducted to offer a better assessment of different decision heuristics for zChaff and to assess a straightforward distributed algorithm for bounded model checking. The results provide a good perspective on the importance of the IBM Formal Verification Benchmark.

3.2.1 Experimental Settings

We ran RuleBase on the IBM benchmark with several configurations on 867841X Intel(R) Xeon(TM) CPU 2.40GHz blade workstations running Red Hat Linux release 7.3. When RuleBase was used with engines distributed on several workstations, they were interconnected with a 1 Gb Ethernet LAN. The different configurations are as follows:

Sequential: RuleBase runs SAT bounded model checking on a single workstation.

The bounds used sequentially include k=0. . . 10, k=11. . . 15,. . . , k=46. . . 50. As soon as a satisfiable assignment is found, the search is over. If no satisfiable assignment is found for any of the bounds, the result is then unsat with bound 50.

Concur 7: RuleBase distributes the tasks corresponding to the seven first bounds to seven workstations (i.e., BMC translation and SAT search for k=0. . . 10, k=11. . . 15,. . . , k=36. . . 40). As soon as a satisfiable assignment is found, the whole search is over. If a task finishes without finding a satisfiable assignment, the next task is then assigned to the now idling workstation (until there are no tasks left, in which case the result is unsat 50).

Concur 5: This uses the same principle as Concur 7, but distributes tasks in a five by five manner to five workstations.

Concur 3: This uses the same principle as Concur 7, but distributes tasks in a three by three manner to three workstations.

Concur 5, 5 heuristics: RuleBase runs five independent SAT BMC instances (similar to sequential). Each of the SAT BMC instances uses a different decision heuristic for zChaff, from VSIDS, SO, SB, DV, DVB.

Concur 9 2/nodes: RuleBase distributes the nine SAT BMC tasks (corresponding to bounds k=0. . . 10, k=11. . . 15,. . . , k=46. . . 50) on five bi-processor workstation.

Table 3.3 presents the experimental results for the rules from the IBM Benchmark. We omitted the rules that ran in under two minutes with Sequential configuration and the rules that timed out (timeout was set at two and a half hours) for every configuration. For this experiment, we used bi-processor workstations with 867841X Intel(R) Xeon(TM) CPUs 2.40GHz and Red Hat Linux release 7.3. Speedup NA means the search timed out for both configurations (i.e., Sequential).

	result	S	C7	C5	H5	C9/2	C3
		(hh:mm:ss)	(speedup)	(speedup)	(speedup)	(speedup)	(speedup)
02_1 rule 1	Unsat 50	00:06:00	2.0	1.9	1.5	2.0	1.6
02_1 rule 2	Unsat 50	00:08:19	2.8	2.7	1.5	2.8	2.2
02_3 rule 2	Unsat 50	00:09:02	2.7	2.6	1.4	2.7	2.5
02_3 rule 4	Unsat 50	00:09:46	2.4	2.3	1.2	2.4	1.9
02_3 rule 6	Unsat 50	00:09:25	2.3	2.2	1.7	2.4	1.8
02_3 rule 7	Unsat 50	00:05:18	2.0	1.9	1.1	2.1	1.7
06	Sat 31	00:02:25	1.8	1.7	1.1	1.8	1.6
10	Unsat 50	00:29:47	2.7	2.6	1.9	2.8	2.0
11 rule 1	Sat 31	00:25:26	8.9	8.2	1.3	8.8	7.4
14 rule 1	Unsat 50	00:02:10	2.3	2.2	1.0	2.7	1.7
14 rule 2	Unsat 50	00:25:43	2.9	2.7	1.0	1.9	2.2
17_1 rule 2	pass	00:05:13	2.5	2.4	1.0	2.1	1.9
18	Sat 29	00:38:59	1.4	1.3	1.3	1.1	1.3
19	Sat 29	00:02:23	1.4	1.4	1.1	1.2	1.2
20	Sat 44	>02:30:00	1.9	1.8	NA	1.8	1.3
22	Unsat 50	>02:30:00	NA	NA	1.6	NA	NA
23	Sat 36	>02:30:00	6.8	6.8	NA	6.3	5.9
26	Unsat 50	00:15:25	3.4	3.1	1.0	3.8	2.3
29	Sat 26	>02:30:00	26.5	26.6	NA	21.8	9.7

Table 3.3. Experimental results with the IBM Formal Verification Benchmark. S stands for Sequential, C7 for Concur 7, C5 for Concur 5, C3 for Concur 3, H5 for Concur 5 with five heuristics, C 9/2 for Concur 9 with two processes per node. and the given configuration).

3.2.2 Interpretation of Results

We noticed that Concur 9 2/node performs more poorly than Concur 5. This may seem quite surprising at first. However, we should keep in mind that BMC translation and SAT solving are very memory accesses consuming. Therefore, memory access can be a bottleneck when running two SAT instances on bi-processor workstations. Heuristic tuning sometimes allows spectacular speedup for SAT solving (the only way we were able to achieve a result for 22). However, the overall improvement for SAT BMC, even when concurrently running several heuristics, is altogether marginal (though this approach could be mixed with the Concur approach). Concur 7 and Concur 5 produce results that are similar, however, the Concur 3 results are significantly poorer. Therefore, the ideal configuration for our search (incremental bounded model checking with a maximal bound of 50) appears to be Concur 5.

Concur k (*i. e.* Concur 3, Concur 5, Concur 7) configurations may give more than linear[3] speedup (e.g., 11 rule 1 and 29). The difficulty of a SAT BMC search does not necessarily grow with the bound. For some rules, it is far more difficult to prove they are no satisfiable assignment for k-5 than to find a satisfiable assignment for k, when searching concurrently. On the other

hand, for some models, if k is the smallest bound for which a satisfiable assignment can be found, it will be easier to find a satisfiable agreement for k+5 than for k. Because k is the length of the shortest counter-examples to the model, it is likely that there will be more counter-examples of a longer length, eg k+5, and so more satisfiable assignments. In summary, Concur k configurations have the biggest speedup potential for models that fail (in a number of cycles less than the maximal bound). Most such models from our benchmark fail in less than 30 cycles. This explains why Concur 7 displays little improvement when compared to Concur 5. In order to exhibit better performance for Concur k with k greater than or equal to 7, we would have to run the SAT BMC with a greater maximal bound and probably a broader benchmark (with models failing within a greater number of cycles). As can be observed in

	Concur 7	Concur 5	Concur3	5 heuristics
Global speedup (Sequential total time/total time)	>3.40	>3.40	>2.78	>1.14
Average speedup (average Sequential time/time)	>4.26	>4.13	>2.49	>1.29

Table 3.4. The results are computed without the "NA" cases

Table 3.4, the Concur 5 and Concur 3 results are quite good, especially considering that these two configurations distribute their tasks in a straightforward manner on five and three nodes, respectively.

3.3 Conclusions

We explained how we used the IBM Formal Verification Benchmark to prove that previous results on SAT tuning were too optimistic and to assess the efficiency of a straightforward distribution for day-to-day SAT bounded model checking. We noted that a wider benchmark would allow even better and sounder assessments. As a result, we plan to make the IBM Formal Verification Benchmark library a living repository. We will add new design models in the future, to increase the benchmark diversity and keep it relevant in light of new technological advances.

3.4 The IBM Formal Verification Benchmark

With the increased use of formal verification, benchmarking new verification algorithms and tools against real-life test-cases is now a must in order to assess performance gains. However, industrial designs are generally highly proprietary; therefore, models generated from these designs are usually not published. This makes difficult to assess the results reported in papers from the industry, as their benchmarks are usually not available and it is not possible to compare the published results with those achieved by other engines. Additionally, formal verification algorithms described in academic papers are often difficult to assess in terms of performance, since they are usually not applied to "real-life" benchmarks. We want to stress how difficult it is to assess the real practical value of more sophisticate theoretical results without proper benchmarking. In the past, benchmarking enabled significant technology improvements, such as those for BDD packages in [**Ya**]. In the same way, many major improvements to boolean satisfyability solvers were proven useful by experimental results [**ZM**]. This may appear trivial, however there are many examples in the literature where elaborated and complex algorithms are not evaluated in a satisfactory manner. From the author's experience, the claims of several papers could not be reproduced using the IBM Formal Verification Benchmark.

The IBM Formal Verification Benchmark library encompasses 37 declassified models, from 31 different hardware designs. The IBM Formal Verification benchmark library is available for academic users from the IBM Haifa verification projects web site [**IFVBL**].

The designs presented in the library are industrial designs that were verified by IBM teams. Each of the benchmark's 37 files contains:

A group of one or more temporal formulas, collectively called "rule". To avoid language compatibility issues related to the specification language, the original PSL/Sugar [**Be**,**Acc**] formulas were translated into very simple $G(p)$ formulas (still written in PSL/Sugar), which most model checkers can readily address.

A design model in PSL/Sugar environment description layer format. Some variables were renamed and some simple reductions were applied to hide the original design intent.

The models presented are of different sizes (Cf. Table 3.5) and varying degrees of complexity. However, as shown in Table 3.6 and Table 3.3, the same problem can sometimes be easily solved by one verification engine and at the same time with difficulty for another engine. For this reason, we tried to use a variety of problems. This benchmark is available in PSL/Sugar [**Acc**] and in Sugar1 format [**Be2**]. It was also translated to BLIF[**BLIF**] format. The CNF output of BMC, applied to the benchmark for several bounds, is available from [**ICB**]. Some of these CNFs were used for the SAT2003 and SAT2004 contests[**SAT03**,**SAT04**].

Name	Variables	Gates	Formulas	Name	Variables	Gates	Formulas
IBM 01	94	3266	1	IBM 17-1	1582	29190	2
IBM 02-1	139	1699	5	IBM 17-2	1581	28807	2
IBM 02-2	135	1671	1	IBM 18	78	4768	1
IBM 02-3	177	1983	7	IBM 19	120	5557	1
IBM 03	109	2656	1	IBM 20	78	4805	1
IBM 04	222	5067	1	IBM 21	78	4768	1
IBM 05	309	8410	1	IBM 22	103	6451	1
IBM 06	132	3375	1	IBM 23	102	6259	1
IBM 07	438	1341	1	IBM 24-1	49048	125896	3
IBM 08	395	84886	1	IBM 24-2	44807	115151	2
IBM 09	232	2000	1	IBM 25	120	4501	1
IBM 10	218	8702	6	IBM 26	1713	9640	1
IBM 11	222	8987	3	IBM 27	42	999	1
IBM 12	224	1055	1	IBM 28	95	3303	1
IBM 13	1506	17459	27	IBM 29	90	2562	1
IBM 14	156	3066	2	IBM 30	180	6654	1
IBM 15	231	4884	1	IBM 31-1	224	2488	3
IBM 16-1	1163	21750	1	IBM 31-2	224	2488	2
IBM 16-2	1162	21674	6				

Table 3.5. IBM Formal Verification Benchmark Circuits Details

We present some sample results[4] achieved against this benchmark. To achieve these results, we used the Discovery engine, one of the RuleBase Classic [**Be1**] BDD engines. Each rule was run "from scratch" (i.e., without taking advantage of any pre-existing BDD orders). In real-life projects, the Discovery engine runs considerably faster when a good BDD order was found previously. This occurs because RuleBase can take advantage of rules previously run for this design, and get the best BDD order for a new rule. However, since the notion of "good order" is not very precise, and because an accurate description of such an order would comprise its entire

listing, we only present results of runs without pre-existing orders. In fact, RuleBase includes a set of engines of significantly higher performance than those referenced here.

We used an IBM Cascades PC with a Pentium III 700 MHz microprocessor running Red Hat Linux Advanced Server Release 2.1AS (Pensacola). The results are presented in Table 3.6 in an hh:mm:ss format.

Name	Discovery	Name	Discovery
IBM 01	0:04:29	IBM 17-1	timeout
IBM 02-1	0:02:04	IBM 17-2	timeout
IBM 02-2	0:00:30	IBM 18	0:02:49
IBM 02-3	0:01:23	IBM 19	0:05:46
IBM 03	0:01:26	IBM 20	0:08:35
IBM 04	0:05:25	IBM 21	0:04:14
IBM 05	0:19:09	IBM 22	1:40:22
IBM 06	0:07:31	IBM 23	0:09:32
IBM 07	0:01:05	IBM 24-1	timeout
IBM 08	0:23:38	IBM 24-2	timeout
IBM 09	0:00:06	IBM 25	timeout
IBM 10	2:48:16	IBM 26	timeout
IBM 11	1:32:45	IBM 27	0:00:18
IBM 12	timeout	IBM 28	1:34:52
IBM 13	0:03:44	IBM 29	0:15:18
IBM 14	timeout	IBM 30	timeout
IBM 15	3:08:01	IBM 31-1	timeout
IBM 16-1	timeout	IBM 31-2	timeout
IBM 16-2	0:08:15		

Table 3.6. Experimental Results – time out: 3 hours

4 Future activity: Investigating Incrementality

States of the FSM are given by the assignment of truth values to the *state variables*. The FSM has a non-empty set of *initial states*, and the *reachable states* are all those states which can be reached from one of these initial states. A safety property is specified as a propositional formula over the state variables. The aim is to prove that the safety property holds in every one of the reachable states.Transitions of the FSM are represented by a propositional formula $T(s, s')$ and the set of initial states by a formula $I(s)$. The safety property that is to be proved is denoted by $P(s)$, and the value of the state variables at time n are denoted by s_n. The shorthand notations I_n, P_n and T_n are used in place of $I(s_n)$, $P(s_n)$ and $T(s_n, s_{n+1})$ respectively. Furthermore, $[\varphi]^p$ is used to denote a set of clauses defining φ, such that p is the literal representing the truth-value of the whole formula. To this end, p is called the *definition literal* of φ. The expression $[\varphi]$ is used as shorthand for $[\varphi]^p \cup \{p\}$.

4.1 Incremental SAT

Standard DPLL solvers which use conflict analysis and clause reordering techniques traditionally take a complete propositional formula and state whether or not it is satisfiable. If there are a number of similar SAT instances to be solved, then the solver will potentially carry out a high number of the same inferences for each one. Incremental SAT addresses this issue by allowing new clauses to be added to the database and the solver run again, without starting the search process from the beginning. The motivation behind this functionality is that learned clauses may not only be useful in that particular problem, but in *similar* ones too. This extension is still quite restrictive as there is no provision for the removal of clauses from the database, a facility which greatly increases the range of problems that can be tackled.

A method to remove, as well as add, clauses to a modern state-of-the-art DPLL based solver (one which uses *conflict clauses* for learning purposes) was suggested by [WKS01]. However, when clauses are to be removed, deciding which conflict clauses also have to be removed requires considerable analysis, and thus time. The reason being that the conflict clauses are dependent on other clauses in the database - removing one or more clauses may therefore invalidate some of the conflict clauses.

To avoid this problem Eén and Sörensson [ES03a] proposed that clauses should only be able to be added but that when the solver is called it is passed a list of unit literals which are assumed to be true. These are then "forgotten" when the solver returns. The advantage of this method is that *all* the learned clauses can be kept in the database since conflict clauses are independent of the assumptions under which they are made.

Note, also, that the removal of added clauses can be achieved in the following manner:

1. augment each clause to be added with a new variable.
2. call the solver with that variable set to false.
3. to delete the clauses, call the solver with the new variable set to true.

Although this will introduce a significant number of new variables, they will all be passed as unit clauses to the solver, so will be removed by the simplification procedures before the search begins.

addClauses(**Initial**) -- *the initial clause set*

for $i \in 0..n$ **do** -- *for each problem instance*
  addClauses($[\mathbf{P}_n]^{p_n}$) -- *add clauses specific to the current problem*
  solve(**LiteralList**) -- *solve current problem (passing necessary assumptions)*

Any clauses which may need to be removed at some stage should be augmented with a definition variable, as described above. The list of assumption literals passed to the solver are used to determine which clauses to keep and which to discard for that specific problem instance.
This interface has been implemented in the MINISAT [ES03b] boolean satisfiability solver, available from http://www.cs.chalmers.se/~een.

4.2 BMC and Incremental BMC

Bounded Model Checking (BMC) was introduced at the end of the last decade [BCCZ99] as an alternative to Binary Decision Diagrams (BDDs) for showing the presence of, or proving the absence of specific properties in a given system. For successively increasing problem bounds (*n*), a formula which encodes the statement "*the property P holds for all paths of length n which begin in an initial state"* is generated. This formula is then passed to a SAT solver which states whether it is satisfiable or not. The problem bound is increased until either a maximum value is reached, or the property is found to be false. Because of the nature of BMC, unless the upper bound on possible bug lengths is known, it is not possible to prove that a property holds for *any* value of *n*, just that it holds for all paths of length *no greater* than *n*.

Typically, for a given value of *n*, the formula which is generated is similar to the one for *n-1*, which in turn is similar to the one for *n-2*, and so on. Each individual formula is treated as a completely new problem instance by the SAT solver, instead of it exploiting these similarities. Bounded Model Checking therefore has a great deal to gain from the incremental techniques described in the previous section.

For increasing values of *n* (starting at 0), the following information needs to be represented by the formula which is to be solved:

- the property to be checked should hold in all states reachable in *n* steps from an initial state.
- each path must not contain any duplicated states (equivalent to forcing every state to be unique).

Clauses which represent the fact that the property has to hold in all states reachable in *n-1* steps can be removed, as they are no longer needed for the subsequent stages. The uniqueness requirement can be encoded by including extra clauses over and above those which state that paths of length *n-1* must have no loops. This form of Incremental Bounded Model Checking is equivalent to just extending the base-case in Temporal Induction (see the next section). An upper bound on possible bug lengths can be found using Temporal Induction techniques too. This operates in a similar manner to what is described here except that it works backwards, finding the longest distinct state paths in which the property eventually does not hold.

4.3. Incremental Temporal Induction

Temporal Induction was introduced in [SSS00] as a method for reasoning about safety properties

over the individual time steps of a FSM. A modification to this algorithm that operated incrementally was later presented in [ES03a]. As with a standard induction proof, a temporal induction proof consists of a base-case and an induction-step. For a given number of steps, the base-case proves that the property always holds, while the induction-step proves that no state where the property is false can be reached with one more transition. The base-case and induction-step are defined in terms of the following formula:

$$\mathbf{Base}_n := \quad \mathbf{I}_0 \wedge \big((\mathbf{P}_0 \wedge \mathbf{T}_0) \wedge \ldots \wedge (\mathbf{P}_{n-1} \wedge \mathbf{T}_{n-1})\big) \wedge \neg\mathbf{P}_n$$
$$\mathbf{Step}_n := \quad \big((\mathbf{P}_0 \wedge \mathbf{T}_0) \wedge \ldots \wedge (\mathbf{P}_n \wedge \mathbf{T}_n)\big) \wedge \neg\mathbf{P}_{n+1}$$

4.3.1. Basic Temporal Induction

Base-Case

The base-case is defined as all paths of length n starting from an initial state such that **P**, the property to be checked, holds in all of the states *except* the nth one. Therefore, if the base case is unsatisfied then the property holds for all paths of length n, starting from an initial state (when checking a base case, it is assumed that all shorter base cases have already been proven). In other words, assuming that the property holds for the first $n-1$ states, there is no path of length n to a state in which **P** does not hold.

Induction-Step

The induction step is defined as a path of length $n+1$ where **P** holds in all states *except* the last one. Once again, if it is unsatisfiable then, given that the property holds for a path of length n, there exists no next state where it does not hold. The states in the induction step must be unique to ensure that the method is complete. Without this restriction it may not be provable even though the property is true. The reason being that if a path contains a loop then it is infinite, thus the induction-step will hold for *any* length, even if the property is eventually false.

Algorithm

The following algorithm checks successively increasing lengths of base and step cases.

for $n \in 0..\infty$ **do**
  if (*satisfiable*([$\mathbf{Base}_n$]))
    return PROPERTY FAILS
  if ($\neg$*satisfiable*([$\mathbf{Step}_n$] $\cup$ [$\mathbf{Unique}_n$]))
    return PROPERTY HOLDS

where $\mathbf{Unique}_n$ is a set of constraints representing the fact that every state must be different.

This algorithm can be modified in a number of ways, for example checking only the base-case gives a pure bug hunting algorithm which will find counter-examples more quickly (it cannot however prove that no bug exists). Also, executing body of the **for** ... **do** loop for every value of n may take too long, especially if the bug or proof is deep. Therefore, starting at a higher n and taking bigger steps can reduce the time taken to return a result. This may not return the shortest counter-example though, whereas the algorithm given above is guaranteed to do so.

4.3.2. Exploiting Incremental Induction

The temporal induction algorithm presented in the previous section can be split in to two parts -- one to find increasing lengths of base-cases and the other to find increasing lengths of induction-steps. The incremental algorithms for these two parts and an interleaving of them, all given in [ES03a], are shown and described below.

Extending the Base-Case

```
addClauses([I_0])
for n ∈ 0..∞ do
    addClauses([P_n]^{p_n})
    solve({¬p_n})
    if (SATISFIABLE)
        return PROPERTY FAILS
    addClause({p_n})
    addClauses([T_n])
```

The first line adds the formula which represents the set of initial states. Line 3 adds the clauses representing the safety property for the states reachable in n steps ($\mathbf{P}_n$) -- this is done because it generally makes the SAT problem easier. These clauses are augmented with an additional new variable that is assumed to be false when the solver is called. If the property holds then a unit clause containing only the negation of the new literal is added to the database, thus resulting in the removal of the previously added clauses. Finally, clauses representing the transition to the next state are added and the process (from line 2) is repeated.

This algorithm terminates if the property is found not to hold, or once a desired value for n has been reached (by replacing ∞ with the desired maximum value).

Extending the Incremental-Step

```
addClauses([¬P_0])
for n ∈ -1..-∞ do
    solve({})
    if (UNSATISFIABLE)
        return INDUCTIVE STEP HOLDS
    addClauses([T_n])
    addClauses([P_n])
for i ∈ 0..n+1 do
    addClauses([s_i ≠ s_n])
```

This algorithm works backwards from the states in which the property does not hold. At any stage of the algorithm, if the clause set is satisfiable then there is a route consisting of $n+1$ steps that leads to a state where the property does not hold. In each of the initial n steps, the property is assumed to hold. When the formula is unsatisfiable, this means that there are no paths of length greater than n (where each state is unique) that lead to a state in which the property does not hold. Therefore, the longest possible counter-example will be of length n. At each stage, when the step is being extended, the clauses representing the transitions to the previous states are added, as well as those which say that the property should hold in every such state. The restriction to unique states ensures that the paths considered do not contain any loops and are therefore finite.

Combined Algorithm

There exist many possible ways in which to combine algorithms 2 and 3. The following is an example of how they can be interleaved, thereby allowing the solver to share conflict clauses between the two processes.

```
addClauses([I_0]^z)              -- z is the definition literal for I_0
for n ∈ 0..∞ do
    addClauses([P_n]^p_n)            -- p_n is the definition literal for P_n
    solve({¬p_n})                    -- induction-step: I_0 not included
    if (UNSATISFIABLE)               -- property guaranteed to hold
        return PROPERTY HOLDS
    solve({z, ¬p_n})                 -- base-case: include I_0
    if (SATISFIABLE)             -- counter-example found
        return PROPERTY FAILS
    addClause({p_n})             -- assert P_n from now on
    addClauses([T_n])            -- assert transition from s_n to s_n+1
for i ∈ 0..n-1 do                -- add uniqueness constraints
    addClauses([s_i ≠ s_n])
```

The definition literals are used to control how their clauses are used by the solver. Not asserting the definition literal at all is equivalent to not including those clauses in the overall formulae, since they are tautological. Note that such clauses are satisfied when either the original variables satisfy them and the definition literal is true, or the original variables do not satisfy them and the definition literal is false. In both cases the definition literal will take on the required truth assignment. Asserting it to be true forces them to be satisfied by the original variables only and asserting it to be false forces them to be unsatisfied by the original variables.

If the induction-step is unsatisfiable then the property is guaranteed to hold because a base-case of length n has already been proven, therefore any counter-examples are of greater length. However, recall that if the induction-step is unsatisfiable, then this means that the longest possible counter-example is of length n therefore the property holds as there are no possible counter-examples.

4.3.3. A graphical Notation

It is possible to have a pictorial representation of thr algorithms described in previous section. In the following, the term *bad state* is used to mean any state in which the safety property does not hold. Counter-examples are symmetric with respect to the initial states and the bad states, that is they begin in an initial state, have a number of intermediate states and end in a bad state. As such, if the transition relation was inverted and the initial and bad states swapped then everything described so far could have been carried out in reverse.

The induction-step is therefore a method for finding an upper bound for the counter-example, and the base-case as a way to produce the counter-example. Graphically, shortest counter-examples will look like the following (where **B'** and **I'** are used to mean **¬B** and **¬I** respectively):

length 0: **IB**

length 1: **IB' ←T→ I'B**

length 2:	**IB'** ←T→ **I'B'** ←T→ **I'B**
length 3:	**IB'** ←T→ **I'B'** ←T→ **I'B'** ←T→ **I'B**
	...
length n:	**IB'** ←T→ **I'B'** ←T→ ... ←T→ **I'B'** ←T→ **I'B**

As can be seen from the diagram, there is a significant amount of sharing between the counter-examples of different lengths. More specifically, if the initial **I** or the final **B** is removed from the nth counter-example:

(1)	**B'** ←T→ **I'B'** ←T→ ... ←T→ **I'B'** ←T→ **I'B**
or (2)	**IB'** ←T→ **I'B'** ←T→ ... ←T→ **I'B'** ←T→ **I'**

then *any* counter-example of length *n* or longer will include both these sections. This means that if the clauses representing (1) and (2) are unsatisfiable then *all* shortest counter-examples of longer lengths will also be unsatisfiable. Therefore this gives an upper bound on the length of the shortest counter example.

Since it is a *shortest* counter-example that is being considered, it can also be concluded that:

1. Between no two states is there a shorter path

or weaker 2. Between no two non-neighbours is there a transition (and the last state is unique)

or weaker 3. No two states are the same

The weaker versions are easier to implement, and using only the third condition is enough to make the algorithm complete.

4.4. Future Steps

The algorithms described in this section are going to be integrated within the NuSMV [CCGPR02] model checker. Then, we will experiment with the above algorithms on the IBM benchmarks that have been used and described in previous sections (The EDL version of the benchmarks can be retrieved at

http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/fvbenchmarks.html).

So far we achieved the following results:

- we translated the EDL version of the IBM benchmarks to the SMV language needed to be given in input to the NuSMV model checker.
- We started the integration of the algorithms described in this document within the NuSMV model checker.

We will provide at month +10 from the beginning of the PROSYD project an revised version of this deliverable, extended with the results of the experimental analysis.

5 References

[Be00] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison Wesley Longman, 2000

[KV97] Orna Kupferman, Moshe Vardi, *Modular Model Checking*, in Proceedings of the COMPOS'97 International Simposium, Lecture Notes in Computer Science, Springer, 1997.

The Quest for Efficient Boolean Satisfiability Solvers, Lintao Zhang, Sharad Malik
New Features of the SAT'04 versions of zChaff , Zhaohui Fu Yogesh Mahajan Sharad Malik

BerkMin: a Fast and Robust Sat-Solver, Evgueni Goldberg Yakov Novikov

SAT and ATPG: Boolean engines for formal hardware verification, Armin Biere, Wolfgang Kunz, Dept. of Computer Science Dept. of Electrical Engineering, ETH, Zürich, Switzerland University of Kaiserslautern, Germany

Zchaff- ://www.ee.princeton.edu/~chaff , Yogesh Mahajan, Zhaohui Fu

[CCGPR02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella "NuSMV 2: An OpenSource Tool for Symbolic Model Checking" In Proceeding of International Conference on Computer-Aided Verification (CAV 2002). Copenhagen, Denmark, July 27-31, 2002

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke and Y. Zhu "Symbolic Model Checking Without BDDs" in TACAS 1999, LNCS:1579, Springer.

[ES03a] N. Eén, N. Sörensson "Temporal Induction by Incremental SAT Solving" First International Workshop on Bounded Model Checking, 2003. ENTCS issue 4 volume 89.

[ES03b] N. Eén, N. Sörensson "An Extensible SAT-solver" short version to appear in SAT2003 formal proceedings. Full paper available from http://www.cs.chalmers.se/~een/Satzoo/An_Extensible_SAT-solver.ps.gz.

[SSS00] M. Sheeran, S. Singh, G Stålmarck "Checking Safety Properties Using Induction and a SAT-solver" in FMCAD 2000, LNCS:1954.

[WKS01] J. Whittemore, J. Kim, K. Sakallah "SATIRE: a new incremental satisfiability engine" in Proc. of the 38th Conference on Design Automation, pages 542 - 545, ACM Press 2001.

The Siege satisfiability solver: http://www.cs.sfu.ca/~loryan/personal/

[Acc] Accelera. PSL/Sugar LRM. http://www.eda.org/vfv/

[Be1] I. Beer *et al.* "RuleBase: An Industry Oriented Formal Verification Tool". In 33rd Design Automation Conference, DAC 96. ACM/IEEE, 1996.

[Be2] I. Beer *et al*. "The Temporal Logic Sugar". In Computer Aided Verification, Proceedings of the 13th International Conference, CAV 2001. LNCS 2102, July 2001.

[Bi2] A. Biere *et al.* « Bounded Model Checking". In Vol. 58 of Advances in Computers. Academic Press (pre-print), 2003.

[Co] S. Cook. "The Complexity of Theorem Proving Procedures". In Proceeding, Third Annual ACM Symp. on the Theory of Computing, 1971.

[DLL] M. Davis, G. Logemann, and D. Loveland. "A machine program for theorem proving". In Journal of the ACM, 5(7), 1962.

[Mo] M. Moskewicz *et al.* « Chaff: Engineering an Efficient SAT Solver". In 38th Design Automation Conference, page 530-535. ACM/IEEE, 2001.

[Pn] A. Pnueli. "A Temporal Logic of Concurrent Programs". In Theoretical Computer Science, Vol 13, pp 45-60, 1981.

[SZ] O. Shacham, E. Zarpas. "Tuning the VSIDS Decision Heuristic for Bounded Model Checking". In Proceeding of the 4th International Workshop on Microprocessor, Test and Verification, IEEE Computer Society, Austin, Mai 2003.

[SS] J. Silva, K. Sakallah. "Grasp - a New Search Algorithm for Satisfiability". In Technical Report TR-CSE-292996, University of Michigan, 1996.

[St] O. Strichman. "Tuning SAT Checkers for Bounded Model Checking". In Computer-Aided Verification: 12th International Conference, Lecture Notes in Computer Science, 1855. Springer-Verlag, 2000.

[Ya] B. Yang *et al.* « A Performance Study of BDD-Bases Model Checking". In Formal Methods in Computer-Aided Design: 2nd International Conference, Lecture Notes in Computer Science, 1522. Springer-Verlag, 1998.

[ZM] L. Zhang and S. Malik. "The Quest for Efficient Boolean Satisfiability Solvers". In Proceedings of 14th Conference on Computer Aided Verification (CAV2002), Copenhagen, Denmark, July 2002

[BLIF] Berkley Logic Interechange Format (BLIF). University of California, Berkley, 1992. http://www-cad.eecs.berkeley.edu/Respep/Research/vis/usrDoc.html

[IFVBL] IBM Formal Verification Benchmark Library.

http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/fvbenchmarks.html

[ICBL] CNF Benchmarks from IBM Formal Verification Benchmarks Library.
http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html

[ICB] IBM CNF Benchmark Illustration: Berkmin561 vs zChaff.
http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks_illustrations.html

[EDL] Web version of the RuleBase User Manual.
http://www.haifa.il.ibm.com/projects/verification_RB_Homepage/

[SAT03] SAT2003 contest. http://www.satlive.org/SATCompetition/2003/index.jsp

[SAT04] SAT2004 contest. http://satlive.org/SATCompetition/2004/