Annals of Mathematics and Artificial Intelligence (2005) 43: 307–342

The SAT2002 Competition

Laurent Simon [a], Daniel Le Berre [b,*] and Edward A. Hirsch [c,**]

[a] *LRI, U.M.R. CNRS 8623, Université Paris-Sud, 91405 Orsay Cedex, France*
E-mail: simon@lri.fr

[b] *CRIL, F.R.E. CNRS 2499, Faculté Jean Perrin, Université d'Artois, Rue Jean Souvraz SP 18, 62300 Lens Cedex, France*
E-mail: leberre@cril.univ-artois.fr

[c] *Steklov Institute of Mathematics at St. Petersburg, 27 Fontanka, 191023 St. Petersburg, Russia*
http://logic.pdmi.ras.ru/~hirsch/

SAT Competition 2002 held in March–May 2002 in conjunction with SAT 2002 (the Fifth International Symposium on the Theory and Applications of Satisfiability Testing). About 30 solvers and 2300 benchmarks took part in the competition, which required more than 2 CPU years to complete the evaluation. In this report, we give the results of the competition, try to interpret them, and give suggestions for future competitions.

Keywords: Boolean satisfiability (SAT), empirical evaluation

AMS subject classification: 68W20, 03B05

1. Introduction

The SAT2002 solver competition, involving more than 30 solvers and 2300 benchmarks, took place in Cincinnati a decade after the first SAT solver competition held in Paderborn in 1991/1992 [7]. Two other SAT competitions were organized since that date: the Second DIMACS Challenge, held in 1992/1993 [29], and the *Beijing* competition, in 1996.[1] In the last few years, the need for such a competition was more and more obvious, reflecting the recent and important progress in the field. A lot of papers have been published concerning "heuristics" algorithms for the NP-complete satisfiability problem [11], and even software exists that people use on real-world applications. Many techniques are currently available, and it is difficult to compare them. This comparison can hardly be only on the theoretical level, because it often does not tell anything from a practical viewpoint. A competition can lead to some empirical evaluation of current algorithms (as good as possible) and thus can be viewed as a snapshot of the state-of-the-art solvers at a given time. The data collected during this competition

* This work has been supported in part by the IUT de Lens, the Université d'Artois and by the "Région Nord/Pas-de-Calais" under the TACT-TIC project.

** Supported in part by RFBR grant No. 02-01-00089 and by grant No. 1 of the 6th RAS contest-expertise of young scientists projects (1999).

[1] Benchmarks available at `http://www.cirl.uoregon.edu/crawford/beijing/`.

will probably help to identify classes of hard instances, solvers limitations and allow to give appropriate answers in the next few years. Moreover, we think that the idea of such a competition takes place in a more general idea of empirical evaluation of algorithms. In a number of computer science fields, we need more and more knowledge about the behavior of the algorithms we design and about the characteristics of benchmarks. This competition is a step in a better – and crucial – empirical knowledge of SAT algorithms and benchmarks [26,27]. The aim of this paper is to report what organizers learned during this competition (about solvers, benchmarks and the competition itself), and to publish enough data to allow the reader to make is own opinion about the results.

As it was mentioned, in the first half of the last decade, some competitions were organized to compare solvers. However, one major – and recent – step in that area was the creation of the SAT-Ex web site [54], an online collection of results concerning various SAT solvers on some classical benchmarks. Among all the advantages of this kind of publication, SAT-Ex allows to check every solver output, generate dynamically synthesis and add constantly new solvers and benchmarks.

More and more researchers would like to see how their solver compares with other solvers on the current benchmark set. This is not really a problem because only a few CPU days are needed to update SAT-Ex database with a new solver: usually, the new solver will outperform old ones for some series of benchmarks. A problem arises with the introduction of new benchmarks: the benchmarks have to be tested on each solver, and they are likely to give them a hard time. Since all the results available on SAT-Ex were obtained on the same computer (the only way to provide a fair comparison based on the CPU time) it will take ages before seeing results on new benchmarks. To solve that problem, there are several solutions:

– working with a cluster of computers instead of a single one. Laurent Simon is currently working that out, preparing a new (and updated) release of SAT-Ex;
– using SAT-Ex system as a convenient way to take a picture of some SAT solvers efficiency on a given set of benchmarks.

This last point is one of our major technical choices: using SAT-Ex architecture for the competition, providing a *SAT-Ex style* online publication. In order to enlight some of the other choices we made during the competition, let us first recall some SAT statements. Currently, approaches to solve SAT can be divided into two categories: complete and incomplete ones. A complete solver can prove satisfiability as well as unsatisfiability of a boolean formula. On the contrary, an incomplete solver can only prove that a formula is satisfiable, usually by providing a model of the formula (a *certificate* of satisfiability).

Most complete solvers descend from the backtrack search version of the initial resolution-based Davis and Putnam algorithm [14,15], often referred to as DPLL algorithm. It can be viewed as an enumeration of all the truth assignments of a formula, hence if no model is found, the formula is unsatisfiable. Last decade has resulted in many improvements of that algorithm in various aspects both in theory (exponential worst-case upper bounds; the most recent are [13,24]) and in practice (heuristics, especially for k-SAT formulas: [16,17,20,39], data structures [66] and local processing). Forward local processing is used to reduce the search space in collaboration with heuristics (unit propagation lookahead [37], binary clause reasoning [62], equivalence reasoning [38], etc.). Backward local processing tries to correct mistakes made by the heuristics: learning, intelligent backtracking, backjumping, etc. [5,42,65]. Also randomization is used to correct wrong heuristics choices: random ties breaking and rapid restart strategies have been shown successful for solving some structured instances (planning [22], Bounded Model Checking (BMC) [4]).

Another use of randomization is the design of incomplete solvers, where randomness is inherent. There was an increased interest in their experimental study after the papers on greedy algorithms and later WalkSAT [51,52]. Encouraging average-case time complexity results are known for this type of algorithms (see, e.g., [33]). In theory, incomplete solvers could perform (much) better than complete ones just because they belong to a wider computational model. Indeed, there are benchmarks (especially coming from various random generators) on which incomplete solvers perform much better. Worst-case time bounds are also better for incomplete algorithms [50].

A revolution? Furthermore, a completely new approach to solve SAT appeared last year, resulting from the existence of huge SAT instances encoding some specific problems, such as planning [18,31,32] or more recently Bounded Model Checking [1,6,63]. While most of the underlying techniques are not new (DPLL with intelligent backtracking, learning and a rapid restart strategy), one of the main idea was to focus on a carefully engineered solver: when dealing with a huge instance, choosing the right algorithm or data structure is as important as choosing the right heuristics to reduce the search space. Chaff [43,67] was designed from the begining to handle large formulas (more than 100000 variables) from a very specific area (mostly Bounded Model Checking) using "lazy" data structures. Since there is no heuristics shown to be efficient on EDA instances, Chaff also integrated a new form of learning, taking advantage of the overall lazy data structures used: Chaff makes mistakes, but learns quickly! Chaff outperformed existing SAT solvers on Bounded Model Checking instances, and a large set of "structured" (as opposed to random) instances [54]. It looked interesting to establish a new overall picture of SAT kingdom after that "revolution".

Such a competition allows to obtain both new solvers and new benchmarks. It was proved to stimulate the community (more than just by providing *awards* to it). Many breakthroughs in the last years were due to empirical evaluation of algorithms, leading to a better knowledge of algorithms behaviors and of benchmarks hardness. This knowledge allow to propose (and test) new answers. Such a competition can thus be viewed as one of the fundamental part of the research around the topic.

2. Rules and submissions

In order to ensure fairness, all the rules concerning the competition were available a few months before the competition on the web,[2] after public discussions on a SATLive! forum.

The solver and benchmark submission processes were running in parallel. The submitters did not know who else submitted solvers or benchmarks, and what was submitted. All submissions were received and processed by Laurent Simon who kept them in secret from everybody including the two other organizers. After that, he alone (+ system administrators) was running the competition computers. That allowed Edward A. Hirsch to participate in the competition despite of being among the organizers.

2.1. The rules

The general idea was to award the most "generic" solver, i.e. the one that is able to solve the widest range of problems. However, it looked like a nonsense to compare a solver tailored for 3-SAT random instances and one tailored for Electronic Design Automation (EDA) instances, and the same remark applies for complete and incomplete solvers. So we divided the space of SAT experiments into 6 categories: industrial, handmade, random benchmarks for either complete (which could solve both satisfiable and unsatisfiable benchmarks) or all solvers (in the latter case, only satisfiable and "unknown" benchmarks were used, and only satisfiable ones were counted). Submitters were asked to stamp their benchmarks with the correct category.

To rank the solvers in each category, we decided to use the notion of series: a series of benchmarks is a set of closely related benchmarks (for instance, pigeon-hole series consists of hole-2, hole-3, etc. instances). We considered that a series was solved if and only if at least one of the instances of that series was solved. Thus the idea was to award a solver solving a maximum number of series in a given category. To break ties, we decided to count the total number of instances solved. We planned to use CPU time as a last resort but we did not have to use it (note that, besides its effects with the CPU cut-off limit, pure CPU time performances do not play a crucial role in the results: two solvers have the same performance if they solve a benchmark, whatever the exact CPU time it takes). Benchmarks were grouped in series by us, authors were only allowed to submit *families* of benchmarks (a series was one or more families of benchmarks).

Furthermore, if there was a scaling factor between the instances of the series (`hole-2` $\leqslant$ `hole-3` $\leqslant$ `hole-4`, etc.) then we did not launch a solver on the biggest instances if it failed to solve any smaller. The initial idea of this "heuristic" (well-founded in practice on the *pigeon hole* example) was to save CPU time (allowing to discard quickly any weak solver). Later, it happened that this choice had an important impact on results and was not well-founded in general (we will discuss this later in the paper). The scaling information of families of benchmarks was only given by benchmarks author.

[2] `http://www.satlive.org/SATCompetition/cfs.html`.

2.1.1. Input and output formats

We asked submitters to send benchmarks in DIMACS file format.[3] One of the ideas underlying this format is that benchmarks are in CNF and are easy to read (for instance, variables are already indexed by integers). Of course, *generators* of benchmarks were allowed, assuming that authors gave clues for the *interesting* parameters to use with.

The output format (printed by solvers) was detailed in our call for solvers.[4] Briefly, the idea was to allow any solver to print any "comment" line (any information judged as "interesting" by authors) and some special lines for automated interpretation purposes. Information lines are important if one wants to understand results and to be able to interpret the huge amount of data collected during the competition. The output format allows to print the answer (SAT, UNSAT or unknown), and requests a certificate if SAT was claimed. If no answer was (syntactically) found in the output (for instances if the solver crashed or was timed out), then *unknown* was assumed.

2.1.2. Checking results and outputs: What makes a solver buggy

Let us notice a tricky consideration about *buggy* solvers. If SAT was claimed on a satisfiable instance, but the certificate was not correct, then *unknown* (and not *buggy*) was assumed as an answer. Each SAT result is thus certified, and we did not consider as *buggy* a solver that gave a wrong certificate (this can be due for instance to a CPUs exceed while printing the certificate or to a data structure problem if the certificate is displayed on a single line). As a matter of fact, we only considered as "buggy" all solvers that answered incorrectly, UNSAT on a SAT instance (previously known SAT or proved by any other solver during the competition). In addition, solvers are by essence incomplete, because of memory and CPU limitation. Thus, if a solver crashed during the competition (which can be due or not to bugs), we did not consider it as *buggy*. We only considered that its answer was "*unknown*".

Each time a *buggy* solver was found, it was tagged *hors-concours* and discarded from the awards (results were still available "unofficially").

2.1.3. Competition steps

From a practical point of view, the competition ran in several steps, going from March to May 2002. The initial step was exclusively for authors: a machine was opened over the web to allow them to compile/test their sources code in "realistic environment". After that, the competition began:

– *compliance testing*:

  - *solvers*: each of them was compiled and tested on a few benchmarks to check that the solvers conformed input/output requirements of SAT-Ex framework. During that step, some bugs (in the usual sense) were detected and reported to authors. But note that *it was not the aim of that step*. Some fixed version were accepted.

[3] This was more precisely a restriction of this format, as described in our call for benchmarks (`http://www.satlive.org/SATCompetition/cfb.html`).

[4] `http://www.satlive.org/SATCompetition/cfs.html`.

   - *benchmarks*: at the same time, new submitted benchmarks were shuffled (literal renaming, clauses reordering). Comment line were also removed. Some benchmarks were discarded because of incorrect syntactical format.

– *first round*: all solvers ran on all "correct" benchmarks during 40 minutes (see section 2.4 for the computer description). We first ran all the solvers on industrial benchmarks, then handmade benchmarks and finally randomly generated ones (this last ones were run for 20 minutes only, on faster machines).
  In this step, the launching heuristic was applied, and, according to it, each complete solver was launched on each applicable benchmark one time. Randomized solvers (incomplete or not) were launched 3 times on each applicable benchmarks, on industrial and hand-made benchmarks only. To take these 3 executions into account, the median CPU time was taken and the instance was solved if at least one execution solved it (that means that a randomized solver can solve a particular instance and be charged of the maximum CPU time, if only one of the three launches has succeeded).

- *second round*: the top five solvers ran on a part of the remaining unsolved instances during 6 hours. If a solver returned an incorrect result (typically, UNSAT instead of SAT) in the first round, then it was not qualified for this stage (even "unofficially").

2.2. Benchmark submission

The following benchmarks were submitted to **Industrial** category:

bart, homer from Fadi Aloul. Represent FPGA Switch-Box problems, all instances should imply a lot of symmetries, as it is described in [19]. Bart instances are all satisfiable, Homer instances are unsatisfiable.

cmpadd from Armin Biere. These benchmarks encode the problem of comparing the output of a carry ripple adder with the output of a fast propagate and generate adder. They are all unsatisfiable.

dinphil from Armin Biere. These benchmarks are generated from bounded model checking from the well-known dining philosophers example. The instances have the generic name '`dp i t k cnf`', where '`i`' is the number of philosophers, '`k`' is the model checking bound and '`t`' is '`u`' for unsatisfiable or '`s`' for satisfiable. The model for '`i`' philosophers may reach a bug not faster than in '`i`' steps.

cache, comb, f2clk, fifo8, ip, w08, w10 from Emmanuel Dellacherie (TNI-Valiosys, `http://www.tni-valiosys.com/`, France). All these problems represent 18 industrial model-checking examples and 3 combinational equivalence examples.

bmc1 from Eugene Goldberg. Bounded Model Checking (BMC) examples (76 CNFs, 30% of them are unsatisfiable) encoding formal verification of the open-source Sun PicoJava II (TM) microprocessor. These CNFs were generated by Ken Mcmillan (Cadence Berkeley Labs). The complete description of the benchmarks is given at `http://www-cad.eecs.berkeley.edu/~kenmcmil/satbench.html`.

bmc2 from Eugene Goldberg, suggested by Ken Mcmillan (Cadence Berkeley Labs). This small set of 6 BMC instances encodes testing whether a sequential N-bit counter (file `cntN.cnf`) can reach a final state from an initial state in 2^{N-1} cycles. In the initial state all the bits of the counter are set to 0 and, in the final state, all the bits of the counter are set to 1. All CNFs are satisfiable.

fpga_routing from Eugene Goldberg and Gi-Joon Nam (32 CNFs submitted by E. Goldberg and 6 by G.-J. Nam separately, but all instances were generated by G.-J. Nam). These Boolean SAT problems are constructed by reducing FPGA (Field Programmable Gate Array) detailed routing problems into Boolean SAT. More information on transforming FPGA routing problems into SAT, are available at `http://andante.eecs.umich.edu/sdr/index.html`.

rand_net from Eugene Goldberg. This is a set of *miter* CNFs (all unsatisfiable) produced from randomly generated circuits. To produce a miter, a random circuit is generated first. This circuit is specified by the number of primary input variables (`N`), the number of levels in the circuit (`M`) and the "length" (`K`) of wires connecting gates of the circuit (`K=1` means that the output of a gate may be connected only to the input of a gate of the next level). A circuit consists of AND and OR gates and does not contain inverters. So any circuit implements a monotone function (by adding inverters to a randomly generated circuit one can make it very redundant). Circuits are "rectangular", i.e. the number of primary inputs, the number of gates of mth level, and the number of primary outputs are all equal to `N`. Now, to check if a circuit is equivalent to itself, a miter is formed. This class of benchmarks allow one to vary the "topology" of the circuit by changing the "length" of wires. Each instance is named `rand_netN_M_K.miter.cnf` where `N`, `M` and `K` are the values of parameters described above.

mediator from Steven Prestwich. The encoded problem (described in [47]) is to construct a query plan to supply attributes in a mediator system (e.g., an online bookstore). These problems combine set covering with plan feasibility and involve chains of reasoning that should make them hard for pure local search. Symmetry breaking constraints were not added, in order to make the problems harder for systematic backtrack search. A file `medN.cnf` contains a problem with shortest known plan length `N`.

IBM from Emmanuel Zarpas (IBM). Bounded Model Checking for real hardware formal verification. Benchmarks are partitioned by difficulty in {Easy, Medium, Hard} by the submitter.

The following benchmarks were submitted to **Handmade** category:

lisa from Fadi Aloul. Those instances represent integer factorization problems. They are all satisfiable (see [19]). Note that other factorization problems (given as generators) were submitted (described below).

matrix, polynomial from Chu-Min Li (with Bernard Jurkowiak and Paul W. Purdom). Those instances encode respectively the multiplication of two $n \times n$ matrices using m products, and the multiplication of two polynomials of degree-bound n using m products. Both problems should involve a lot of symmetries (see [8]).

urquhart from Chu-Min Li (with Sebastien Cantarell and Bernard Jurkowiak) [38] and independently from Laurent Simon [10]. All instances are unsatisfiable and proved very hard for all DLL and DP approaches (hard for all resolution-based procedures, in general [57]). Chu-Min Li benchmarks are 3-SAT encoding of Urquhart problems and Laurent Simon are non-reduced encoding (clauses can be long).

hanoi from Eugene Goldberg (but generated by Henry Kautz). These instances represent the classical problem of the Towers of Hanoi, hand-encoded axioms around 1993 (similar to the ones used in [29], but larger instances available).

graphcolor*K* from Dan Pehoushek. Random regular graph coloring problems. Above some number of vertices, most of them should be colorable.

ezfact from Dan Pehoushek. SAT encoding of factorization circuits.

glassy-sat-sel from Federico Ricci-Tersenghi. Selected instances (by the submitter) of medium hardness from the glassy-sat generator (see below).

gridmnbench from Allen Van Gelder. Encode (negated) propositional theorem about a (non realistic) fault-tolerant circuit family.

checkerinterchange from Allen Van Gelder (with Fumiaki Okushi). Planning problem to solve checker interchange problem within deadline.

ropebench from Allen Van Gelder. A linear family of graph coloring problems (sequence of unsatisfiable formulas in 3-CNF). The formula length is linear in the number of variables (namely, $36n$).

qgbench from Hantao Zhang. Small instances of quasigroups with constraints 0–7.

sha from Lintao Zhang and Sharad Malik. CNF encoding of secure hashing problems.

xor-chains (among them, the smallest unsolved unsatisfiable instance with 106 variables, 282 clauses and 844 literals), from Lintao Zhang and Sharad Malik. This encodes verification problems of 2 xor chains.

satex-challenges from Laurent Simon. Selection of (heterogenous) unsolved instances from SAT-Ex [54].

pyhala from Tuomo Pyhàlà. Submitted as a generator. Depending on arguments, it can generate a SAT encoding of factoring of primes (unsat instances) or products of two primes (sat instances). The benchmarks encode multiplication circuits, with predefined output. Two circuits are available (*braun* or *adder-tree* multipliers).

The benchmarks of **Random** category were submitted as generators (except for plainoldcnf and twentyvars):

3sat from the organizers. This generator produces uniform 3-CNF formulas. Checks are performed to prevent duplicate or opposite literals in clauses. In addition, no duplicate clause are created.

glassy-sat from Federico Ricci-Tersenghi (with W. Barthel, A.K. Hartmann, M. Leone, M. Weigt, and R. Zecchina). Generator of hard and solvable 3-SAT instances, corresponding to a glassy model in statistical physics. A description is available as a preprint at `http://xxx.lanl.gov/abs/cond-mat/0111153`.

okrandgen from Oliver Kullman [34,36], k-CNF uniform random generator, based on encryption functions to ensure *strong* and *reliable* random formulae. Detailed description and sources available at `http://cs-svr1.swan.ac.uk/~csoliver/OKgenerator.html`.

hgen2 from Edward A. Hirsch (available from `http://logic.pdmi.ras.ru/~hirsch/benchmarks/hgen2.html`). An instance generated by this generator (3-CNF, 500 variables, 1750 clauses, 5250 literals, seed 1 216 665 065) was the smallest satisfiable benchmark that remained unsolved during the competition. Description: First a satisfying assignment is chosen; then clauses ($3.5n$ of them for n variables) are generated one by one. A literal cannot be put into a clause if

1. There is a less frequent literal.
2. The corresponding variable already appears in the current clause.
3. A variable dependent on it (i.e., occurred together in another clause) already appears in the current clause.
4. A variable dependent on a dependent variable already appears in the current clause.
5. The opposite literal is not satisfying and occurs not more frequently (except for the case that choosing a satisfying literal is our last chance to satisfy this clause).

If the generation process fails (no literal can be chosen), it is restarted from the beginning.

hgen1 from Edward A. Hirsch. Similar to hgen2 except for condition 5.

hgen3 from Edward A. Hirsch. Similar to hgen1, but formulas are *not* required to be satisfiable.

hgen4 from Edward A. Hirsch. Similar to hgen2, but formulas are in 4-CNF, with $9n$ clauses. Also condition 4 is not applied.

hgen5 from Edward A. Hirsch. Similar to hgen2, but formulas are a mix of 3-CNF ($1.775n$ clauses) and 4-CNF ($5.325n$ clauses).

g3 from Mitsuo Motoki. Generates positive instances at random. These instances have only one solution with high probability. Benchmarks were discarded because of a bug in the generator.

plainoldcnf from Dan Pehoushek. Selection of regular random 5-CNF.

twentyvars from Dan Pehoushek. Small instances (in terms of their number of variables) of k-CNF, with $k \in \{6, 7, 8\}$.

2.3. *Solver submission*

We wanted the competition to be as fair and open as possible. So we did not want to restrict people to a given language (such as C or C++): the only condition was that the solver can run on a standard Linux/Unix box. The solver sources had to be provided, with a suitable makefile. Additional libraries were statically linked to the code. All but one solvers were in C/C++, one was in Java.

limmat Armin Biere. Complete deterministic solver. This is a zchaff-like SAT solver (implemented in C) with an early detection of conflicts in the BCP queue; a constant time lookup of the ‘other’ watched literal; an optimized ordering of decision variables and a robust code through sophisticated test framework. More informations and sources are available at `http://www.inf.ethz.ch/personal/biere/projects/limmat/`.

saturn by Steven Prestwich [48]. Incomplete randomized solver.

2clseq by Fahiem Bacchus. Complete deterministic solver. DPLL with binary clause and equivalence reasoning plus intelligent backtracking and learning [2,3]. Available in source (C++) at `http://www.cs.toronto.edu/~fbacchus/2clseq.html`.

marchI, marchIse, marchII, marchIIse by Marijn Heule, Hans van Maaren, Mark Dufour, Joris van Zwieten. Complete deterministic solver. Those solvers were designed by postgraduate students for a course given by Hans van Maaren. The heuristics used in those solvers can be found in [64]. Note that some of them (marchIse-hc, marchII-hc, marchIIse-hc) were received after the deadline so we decided to run them hors-concours.

blindsat by Anatoly Plotnikov and Stas Busygin. Complete deterministic solver. A report and the solver source (C++) are available at `http://www.vinnica.ua/~aplot/current.html`.

ga by Anton Eremeev and Pavel Borisovsky. Incomplete randomized solver. A greedy crossover genetic algorithm.

berkmin by Eugene Goldberg and Yakov Novikov [21]. Complete deterministic solver.

> “Berkmin inherits such features of GRASP, SATO, and Chaff as clause recording, fast BCP, restarts, and conflict clause “aging”. At the same time Berkmin introduces a new decision making procedure and a new procedure for the management of the database of conflict clauses. The key novelty of Berkmin is that this database is organized as a chronologically sorted stack. Berkmin always tries to satisfy the topmost unsatisfied clause of the stack. When removing clauses Berkmin tries to get rid of the clauses that are at the bottom of the stack in the first place” [Eugene Goldberg].

The version used for the competition was 62. Berkmin 56 binaries are available at `http://eigold.tripod.com/`.

unitwalk by Edward A. Hirsch and Arist Kojevnikov [25]. Incomplete randomized solver. UnitWalk is a combination of unit clause elimination (particularly, the idea of Paturi, Pudlák and Zane’s randomized unit clause elimination algorithm [46]) and local search. The solver participated in the competition extends this basic algorithm with adding some of 2-resolvents using incBinSat [68], and mixes its random walks with WalkSAT-like [51] walks. The version used for the competition was 0.98. Available in source (C) at `http://logic.pdmi.ras.ru/~arist/UnitWalk/`.

jquest by Joao Marques-Silva and Inês Lynce. Complete deterministic solver. Jquest is a SAT platform in Java containing various heuristics, data structures and search strategies [40]. The solver was configured with lazy data structures (inspired by both SATO and Chaff), non-chronological backtracking and clause recording (like in Grasp), chaff-like heuristic, randomized backtracking [41] and rapid restarts strategy. JQuest source code is available at `http://sat.inesc.pt/sat/soft/jquest/jquest-src.tgz`.

lsat by Richard Ostrowski, Bertrand Mazure and Lakhdar Sais [45]. Complete deterministic. LSAT detects some boolean functions (equivalence chains, and/or gates) and uses them

- to simplify the original CNF,
- to detect independent variables.

Then a classical DPLL is launched on the simplified CNF, branching only on independent variables.

usat05, usat10 by Bu Dongbo. Incomplete randomized solver. No description available.

sato by Hantao Zhang [65]. Complete deterministic solver.

simo by Armando Tacchella, Enrico Giunchiglia, Marco Maratea [12]. Complete deterministic (wrongly noted randomized in the competition). In Simo3.0 there are features the most recent and effective in SAT like UIP-based learning, restart, 2-literals watching. Simo3.0 is characterized by a new type of heuristic(called GMT). GMT tries to combine Chaff-like and SATZ-like heuristics. The idea is to switch between SATZ-like and Chaff-like heuristics by introducing measures of "probably successful search" and "probably unsuccessful search". The default is to use a Chaff-like heuristic. When the measure of unsuccessful search exceeds a given threshold, SIMO switches to a SATZ-like heuristic. SIMO resumes the Chaff-like heuristic once the measure of successful search exceeds a given threshold. SIMO 2.0 is available in source (C++) at `http://www.mrg.dist.unige.it/~sim/simo/`.

OKsolver by Oliver Kullmann [35]. Complete deterministic solver.

> "OKsolver has been designed to be a "clean solver" as possible, minimising the use of "magical numbers", and for 3-CNF indeed the algorithm is completely generic. OKsolver is a DPLL-like algorithm, with reduction by failed literals (complete and iterated at each node) and autarkies (found when searching for failed literals), while the branching heuristic chooses a variable creating as many new clauses as possible (exploiting full unit clause propagation for all variables), and the first branch is chosen maximising an approximation of the probability, that a branching formula is satisfiable" [Oliver Kullmann].

OKsolver 1.2 source code is available at `http://cs-svr1.swan.ac.uk/~csoliver/`.

dlmsat1, dlmsat2, dlmsat3 by Benjamin Wah and Alan Zhe Wu [53]. Incomplete randomized solvers. Available in source at `http://manip.crhc.uiuc.edu/Wah/programs/SAT_DLM_2000.tar.gz`.

modoc by Allen Van Gelder [44,58,61]. Complete deterministic solver. Binaries available at `ftp://ftp.cse.ucsc.edu/pub/avg/Modoc/`.

rb2cl by Allen Van Gelder [59,60,62]. Complete deterministic solver.

It applies reasoning in the form of certain resolution operations, and identification of equivalent literals. Resolution produces growth in the size of the formula, but within a global quadratic bound; most previous methods avoid operations that produce any growth, and generally do not identify equivalent literals. Computational experience so far suggests that the method does substantially less "guessing" than previously reported algorithms, while keeping a polynomial time bound on the work done between guesses [Allen van Gelder].

zchaff by Lintao Zhang and Sharad Malik [43,67]. Complete deterministic solver. This solver is a carefully engineered DPLL procedure with non-chronological backtracking, learning (clause recording), restarts, randomized branching heuristic and an innovative notion of "heuristic learning" (VSIDS). Zchaff source code is available at `http://www.ee.princeton.edu/~chaff`.

partsat by John Kolen. Complete deterministic solver. No description available.

2.4. Computers available

The competition was held on 2 clusters of Linux PCs, kindly provided by the University of Cincinnati, thanks to John Franco. The first cluster of 32 dual PIII-450 computers with 1 GB of RAM was used to run most of the competition: compliance testing, first stage for handmade and industrial benchmarks, second stage for all benchmarks. The second cluster, consisting of 16 Athlon 1800+ machines, was used to run the first stage on random instances. Only one processor was used, virtual (hard drive) memory was disabled and each program was given 900 MB of memory.

3. The results

The results of the competition were released during the SAT2002 symposium. The detailed results can be found on the competition web page `http://www.satlive.org/SATCompetition/2002/`.

Some of the solvers were found buggy by the organizers during compliance testing, returned to their authors, and corrected (some of them). However, this was not the aim of this phase, and cannot be considered as a guarantee of any kind of testing. Only wrong answers (or obvious crashes) were reported as bugs. We also noticed problems with some solvers during the first round but we did not accept new version of the solvers. Here are some of the things one must be aware of before reading the results.

But, first of all, we must begin with a word of caution: The following results should be considered with care, because they correspond to the behavior of a particular version of each solver (the one that was submitted) on the benchmarks accepted for the competition, on a particular computer under a particular operating system.[5] The competition results should increase our knowledge about solvers, but one inherent risk of such snap-

[5] Linux-SMP 2.4.3, solvers binaries compiled by `gcc 2.96`.

shot is that results can be misinterpreted, and thus lead to wrong pictures of the state of the art.

We discarded some of the solvers from the competition (ran *hors concours*) because they demonstrated unexpected behavior; mainly, claimed UNSAT for a satisfiable instance. Most of the time, the problem showed up only on a few families of benchmarks. Also some versions of the marchXYZ solvers were hors concours from the beginning, because these versions were submitted substantially after the deadline (but before the first stage of the competition). Hors concours solvers results *are* also displayed on the competition web page. For instance, the `lsat` solver answered incorrectly on some instance (there was actually a bug in the code), but the corrected version (as well as the buggy version) solved easily all `urquhart` instances and the `xor-chains` benchmarks, awarded during the competition (adding the detection of boolean functions pays on small but hard hand-made formulas). But it was officially discarded because of its bug.

Some specific problems occured with other solvers (not hors concours). In the following two cases, the picture given by the competition results does not reflect the real solvers performance:

Berkmin was composed of two engines, one for small/medium instances, and one for large instances. The latter just crashed on Linux (the authors tested it under MS Windows and Solaris only). That problem was not detected during the compliance testing (for more details, see `http://www.satlive.org/SATCompetition/2002/berkmin.html`). Note that other solvers also crashed sometimes, especially during the second round where benchmarks were larger.

JQuest did not output a correct certificate when the instance had less variables than the `nbvar` parameter provided in the "`p cnf nbvar nbclause`" line (because in that case, it renames internally the variables ids). For that reason, JQuest is reported not solving those instances.

The best way to view the detailed results of the competition is to take a look at all the traces at `http://www.ececs.uc.edu/sat2002/scripts/menu_choix2.php3`; the summaries of the results per competition stage per category follow. The instances used for the competition are available on SATLIB.[6]

3.1. First stage

In tables 1–3, letfmost number is the number of solved series (a series is solved if at least one of its instances is solved). Rightmost number (where breaking a tie is necessary) denotes the total number of instances solved.

In each category, the top five solvers went to the second stage.

[6] `http://www.satlib.org/`.

Table 1
First stage results on Industrial instances.

Complete solvers on Industrial benchmarks		
23	zchaff	
22	limmat	
18	berkmin	
15	simo	
14	2clseq	
12	jquest	
11	oksolver	
10	rb2cl, march2[se], modoc	
3	blindsat	

All solvers on satisfiable Industrial benchmarks		
11	zchaff	
10	limmat	
8	berkmin	
7	simo	
6	unitwalk	57
6	2clseq	55
6	dlmsat2	54
6	dlmsat1	53
6	rb2cl	50
6	saturn	49
6	oksolver	47
6	march2	44
6	march2se	43
6	jquest	32
5	usat10/usat05, modoc, dlmsat3	
3	blindsat	
1	ga	

3.2. *Second stage*

In this stage, Top 5 solvers were run on smallest remaining unsolved instances for 6 hours. For industrial benchmarks, only 31 instances remained unsolved. So, we used all of them for complete solvers, and only satisfiable instances for all solvers. Thus, in table 4, berkmin, oksolver, 2clseq, simo and zchaff were launched on all instances. unitwalk was only launched on all instances that were not known to be unsatisfiable. Over the 31 instances, only 15 were solved. One can notice that, surprisingly, berkmin and 2clseq are able to solve comb/comb3 instance in less than 2000 s. This benchmark was not solved during the first stage of the competition (recall that CPU cut-off was 2400 s on the same machines), due to the use of our launch-heuristic (comb1 and comb2 are still unsolved, and considered as easiest by the heuristic). Let us also notice how

Table 2
First stage results on Handmade instances.

Complete solvers on Handmade benchmarks		
20	berkmin, oksolver	
19	2clseq, limmat, zchaff	
18	simo	
17	march2se	
16	jquest	
15	rb2cl, march2	
14	modoc	
2	blindsat	
All solvers on satisfiable Handmade benchmarks		
11	berkmin, oksolver, unitwalk	
10	zchaff	73
10	limmat	65
10	2clseq	63
9	dlmsat2, simo, usat05/10	
8	dlmsat3, dlmsat1, jquest, march2se, saturn	
7	march2, rb2cl	
5	modoc	
2	ga, blindsat	

zchaff seems well-tuned for this category: it is able to solve instances with millions of literals.

For handmade instances (table 5), only a few families remained (but several instances per family) so we took the smallest 2 SAT+UNSAT instances in each family for complete solvers, and 2 smallest SAT benchmarks in each family for all solvers (families are urq/urq*bis, urq/urq, xor-chain/x1_*, xor-chain/x1.1_*, xor-chain/x2_*, matrix, Lisa, satex-c/par32-*-c, satex-c/par32, pbu-4, pbs-4 and hanoi). One can notice that the only incomplete solver used in this stage (i.e., uniwalk) was not able to solve any instance during this stage. Let us just recall that most instances in this table are easy for lsat (xor-chains, urq, par32), which was hors-concours.

The selection of random benchmarks was guided by the following considerations: the smallest unsolved unsatisfiable instance had already been found in the handmade category (the smallest unsolved random instance was larger than it). Concerning SAT instances, all the SAT instances in the industrial and handmade categories were tested for the second stage and the smallest unsolved one had 82 345 literals (handmade pbs4 instance). One feature of random category is that most instances are unknown. So, to be sure to award the smallest SAT instance, we needed to keep the smallest instances for the second stage, independently of their series. As a consequence, if all the series were present in that second stage, the number of instances per series varied.

Table 3
First stage results on randomly generated instances.

	Complete solvers on randomly generated benchmarks	
34	2clseq, oksolver	
32	march2, march2se	
31	rb2cl	616
31	simo	569
31	berkmin	541
30	zchaff	
28	limmat, modoc	
17	jquest	
4	blindsat	
	All solvers on satisfiable randomly generated benchmarks	
23	dlmsat1,dlmsat2,dlmsat3	
22	unitwalk	
21	oksolver	261
21	usat10	257
21	saturn	255
21	2clseq	228
20	march2[se], rb2cl, simo, usat05	
19	berkmin	
18	modoc, zchaff	
16	limmat	
9	jquest	
5	ga	
4	blindsat	

Table 6 shows all the results for randomly generated instances, sorted by their respective length. Note that all solved instances were previously known as SAT (by forced-SAT generators).

We finally give as summary of results, in tables 7–9. The leftmost number denotes the total number of instances solved during the second stage. Rightmost numbers (if any) denote the 1st stage result (number of solved series and total number of instances solved to break ties).

3.3. Benchmarks

We awarded the two smallest (one satisfiable and one unsatisfiable) instances that remained unsolved during the competition. Of course, both instances participated in the second stage, i.e., the top 5 solvers were run on them for 6 hours! Note that we did not take into account here the instances that were submitted as “unknown”.

The smallest hard unsatisfiable instance **xor-chain/x1_36** (106 variables, 844 literal occurrences) was submitted by Lintao Zhang and Sharad Malik.

Table 4
Industrial benchmarks used for the second stage. "N*", in the "SAT?" colmun, denotes a previously-unknown benchmark claimed to be Unsat (recall that no proof were given, and solver have to be trusted on this answer). All fpga-r/file are fpga-routing/k2fix_gr_file, 6pipe_o for 6pipe_6_ooo, and satex-c/cnf-r4-i for satex-challenges/cnf-r4-b1-k1.i-comp.

Name	Nb Var	Nb Clauses	Length (Max)	SAT?	Solved by
Homer/homer17	286	1 742	3 718 (12)	N	limmat (6957 s)
Homer/homer18	308	2 030	4 312 (12)	N	
Homer/homer19	330	2 340	4 950 (12)	N	
Homer/homer20	440	4 220	8 800 (12)	N	
dinphil/dp11u10	9 197	25 271	59 561 (12)	N	
dinphil/dp12u11	11 137	30 792	72 531 (13)	N	
comb/comb1	5 910	16 804	38 654 (29)	–	
comb/comb2	31 933	112 462	274 030 (14)	–	
comb/comb3	4 774	16 331	39 495 (14)	N*	berkmin (1025 s), 2clseq (1772 s)
f2clk/f2clk_40	27 568	80 439	186 255 (26)	–	
f2clk/f2clk_50	34 678	101 319	234 655 (26)	–	
fifo8/fifo8_300	194 762	530 713	1 200 865 (12)	N*	zchaff (5716 s)
fifo8/fifo8_400	259 762	707 913	1 601 865 (12)	N*	zchaff (16083 s)
ip/ip36	47 273	153 368	366 122 (21)	N*	limmat (20919 s), zchaff (6982 s)
ip/ip38	49 967	162 142	387 080 (21)	N*	limmat (5640 s), zchaff (13217 s)
ip/ip50	66 131	214 786	512 828 (21)	–	
w08/w08_14	120 367	425 316	1 038 230 (16)	Y	zchaff (16359 s)
w08/w08_15	132 555	469 519	1 146 761 (16)	–	
bmc2/cnt10	20 470	68 561	187 229 (4)	Y	
fpga-r/2pinvar_w8	3 771	270 136	1 620 816 (7)	–	
fpga-r/2pinvar_w9	5 028	307 674	2 438 766 (9)	–	
fpga-r/2pin_w8	9 882	295 998	1 727 100 (7)	–	
fpga-r/2pin_w9	13 176	345 426	2 606 340 (9)	–	
fpga-r/rcs_w8	10 056	271 393	550 328 (9)	–	
satex-c/cnf-r4-1	2 424	14 812	39 764 (25)	Y	limmat (21339 s), berkmin (13071 s)
satex-c/cnf-r4-2	2 424	14 812	39 764 (25)	Y	limmat (20454 s)
fvp-unsat/6pipe	15 800	394 739	1 157 225 (116)	N	zchaff (12714 s)
fvp-unsat/6pipe_o	17 064	545 612	1 608 428 (188)	N	zchaff (4398 s)
fvp-unsat/7pipe	23 910	751 118	2 211 468 (146)	N	
sha/sha1	61 377	255 417	769 041 (5)	Y	
sha/sha2	61 377	255 417	769 041 (5)	Y	

The smallest hard satisfiable instance **hgen2-v500-s1216665065** (500 variables, 5250 literal occurrences) was generated by Edward A. Hirsch's random instance generator hgen2.

Note that instances with fewer variables also remained unsolved, but the winner was determined by the total number of *literal occurrences* in the formula (note that a

Table 5
Handmade benchmarks used for the second stage. pbu and pbs are pyhala-braun-unsat and pyhala-braun-sat, respectively. Comp/Un. denotes which solvers were used: "C" means that we tried berkmin, oksolver, 2clseq, limmat and zchaff on the considered benchmark. "U" means that unitwalk was also launched (see previous section for the Top 5 in Handmade category) and note that berkmin, oksolver, limmat and zchaff where common in both Complete/All categories.

Name	Nb Var	Nb Clauses	Length (Max)	SAT?	Comp/Un.	Solved by
urq/urq3_25bis	99	264	792 (4)	N	C	berkmin (2825 s)
xor-chain/x1_36	106	282	844 (4)	N	C	
xor-chain/x1.1_40	118	314	940 (4)	N	C	
xor-chain/x1_40	118	314	940 (4)	N	C	zchaff (3165 s)
xor-chain/x2_40	118	314	940 (4)	N	C	
xor-chain/x1.1_44	130	346	1 036 (4)	N	C	
xor-chain/x2_44	130	346	1 036 (4)	N	C	
urq/urq3_25	153	408	1 224 (4)	N	C	
urq/urq4_25bis	192	512	1 536 (4)	N	C	
urq/urqu4_25	288	768	2 304 (4)	N	C	
matrix/Mat26	744	2 464	6 432 (4)	N	C	zchaff (18604 s)
satex-c/par32-2-c	1 303	5 206	15 246 (4)	Y	C,U	
satex-c/par32-1-c	1 315	5 254	15 390 (4)	Y	C,U	
Lisa/lisa21_99_a	1 453	7 967	26 577 (23)	Y	C,U	berkmin (20459 s)
satex-c/par32-2	3 176	10 253	27 405 (4)	Y	C,U	
satex-c/par32-1	3 176	10 277	27 501 (4)	Y	C,U	
pbu-4/p-b-u-35-4-03	7 383	24 320	62 950 (4)	N	C	berkmin (3693 s), oksolver (3073 s), 2clseq (10821 s), limmat (4718 s), zchaff (2738 s)
pbs-4/p-b-s-40-4-03	9 638	31 795	82 345 (4)	Y	C,U	
pbs-4/p-b-s-40-4-04	9 638	31 795	82 345 (4)	Y	C,U	zchaff (3182 s)
pbu-4/p-b-u-40-4-01	9 638	31 795	82 345 (4)	N	C	
hanoi/hanoi6	4 968	39 666	98 346 (10)	Y	C,U	berkmin (2551 s)
matrix/Mat317	24 435	85 050	227 610 (4)	–	C,U	

hard randomly generated formula in 4-CNF will have a much greater clauses/variables ratio, not to say about the length of its clauses).

4. Other views of the competition

As we said, one of the risks of such competition is that it results can be misleading (how strong are the results w.r.t. the performance of solvers in a *real* situation: embeded component in a model checker or a planning system for instance?). As long as the competition was running, we had to make decisions, each of them having a direct impact on final results. We have collected a large amount of data, much more valuable than just the name of the final winner. Here, we try to interpret these data from a different point of view. Note that the following is based on the data collected during the first stage of the competition.

Table 6
Random benchmarks used for the second stage. Clause max length is not reported (it is exactly 4 for all benchmarks). Comp is "C" if 2clseq, oksolver, marchII, marchIIse and rb2cl were launched; and "U" if dlmsat [1–3], unitwalk and oksolver were launched (see previous section for Top 5 solvers and random instances).

Name	Nb Var	Nb Cl.	Length	SAT?	Comp/Un.	Solved by
hgen3-v300-s1766565160	300	1 050	3 150	–	C	
hgen3-v300-s1817652174	300	1 050	3 150	–	C	
hgen3-v300-s229883414	300	1 050	3 150	–	C	
hgen3-v350-s1711636364	350	1 225	3 675	–	C	
hgen3-v350-s524562458	350	1 225	3 675	–	C	
hgen2-v400-s161064952	400	1 400	4 200	Y	C,U	unitwalk (20199 s)
hgen3-v400-s344840348	400	1 400	4 200	–	C	
hgen3-v400-s553296708	400	1 400	4 200	–	C	
hgen2-v450-s41511877	450	1 575	4 725	Y	C,U	dlmsat3 (94 s)
hgen3-v450-s432353833	450	1 575	4 725	–	C	
unif-c1700-v400-s734590802	400	1 700	5 100	–	C	
okgen-c1700-v400-s2038016593	400	1 700	5 100	–	C	
hgen2-v500-s1216665065	500	1 750	5 250	Y	C,U	
hgen3-v500-s1349121860	500	1 750	5 250	–	C	
hgen3-v500-s1769527644	500	1 750	5 250	–	C	
hgen3-v500-s1803930514	500	1 750	5 250	–	C	
hgen3-v500-s1920280160	500	1 750	5 250	–	C	
glassybp-v399-s382874052	399	1 862	5 586	Y	C,U	oksolver (13034 s), marchII (7100 s), marchIIse (7064 s)
glassybp-v399-s499089820	399	1 862	5 586	Y	C,U	
glassyb-v399-s500582891	399	1 862	5 586	Y	C,U	oksolver (13834 s)
glassyb-v399-s732524269	399	1 862	5 586	Y	U	oksolver (8558 s)
glassy-v450-s1188040332	450	2 100	6 300	Y	U	oksolver (15444 s)
glassy-v450-s1679149003	450	2 100	6 300	Y	U	oksolver (18766 s)
glassy-v450-s1878038564	450	2 100	6 300	Y	U	
glassy-v450-s2052978189	450	2 100	6 300	Y	U	
glassy-v450-s325799114	450	2 100	6 300	Y	U	
glassybp-v450-s1173211014	450	2 100	6 300	Y	U	
glassybp-v450-s1349090995	450	2 100	6 300	Y	U	
glassybp-v450-s1976869020	450	2 100	6 300	Y	U	
glassybp-v450-s2092286542	450	2 100	6 300	Y	U	
glassybp-v450-s40966008	450	2 100	6 300	Y	U	
glassyb-v450-s1529438294	450	2 100	6 300	Y	U	
glassyb-v450-s1709573704	450	2 100	6 300	Y	U	
glassyb-v450-s1729975696	450	2 100	6 300	Y	U	

4.1. SOTA view

Geoff Sutcliffe and Christian Suttner are running the CASC[7] competition for many years now, and provide in [55] some clues about what is a fair way to evaluate au-

[7] CASC = "CADE ATP System Competition", CADE = "Conference on Automated Deduction", ATP = "Automated Theorem Proving".

Table 7
Second stage results on Industrial benchmarks (summary).

Complete solvers		All solvers on satisfiable benchmarks		
7	**zchaff**	2	**limmat**	
5	limmat	1	zchaff	[11]
2	berkmin	1	berkmin	[8]
1	2clseq	0	simo	[7]
0	simo	0	unitwalk	[6]

Table 8
Second stage results on Handmade benchmarks (summary).

Complete solvers			All solvers on satisfiable benchmarks		
3	**zchaff**		2	**berkmin**	[11]
2	berkmin		2	zchaff	[10]
1	oksolver	[20]	1	oksolver	[11]
1	limmat	[19 140]	1	limmat	[10]
1	2clseq	[19 126]	0	unitwalk	

Table 9
Second stage results on randomly generated benchmarks (summary).

Complete solvers			All solvers on satisfiable benchmarks		
4	**oksolver**		5	**oksolver**	
3	march2, march2se		1	dlmsat3	[23]
0	2clseq	[34]	1	unitwalk	[22]
0	rb2cl	[31]	0	dlmsat1, dlmsat2	

tomated theorem provers. One of the key ideas of their work is the notion of *State Of The Art (SOTA)* solver. It is based on a subsumption relationship between the set of instances solved by the competing solvers: "a solver A is better than a solver B iff solver A solves a strict subset of the instances solved by solver B". The underlying idea is that any solver being the only one to solve an instance is meaningful. The subsumption relationship provides a partial order between the competing solvers, and a virtual solver representing advances of the whole community, *the SOTA solver*. This solver can solve every problem solved by any of the competing solvers (so would be the unique maximal element for the subsumption relationship). There is a little chance that the SOTA solver is a real solver, and thus the notion of *SOTA contributors* is introduced by the authors. They correspond to the maximal elements of the subsumption relationship. The SOTA solver is equivalent to the set of SOTA contributors running in parallel.

Tables 10–12 show the problems solved by only one solver during the first stage of the competition, for the three categories of benchmarks (hors-concours solvers are discarded). We can now find SOTA contributors from each table:

Table 10
Uniquely solved Industrial benchmarks during the first stage.

Solver	Bench (shortname)	CPU (s)
2clseq	bmc2/cnt09.cnf	198.00
2clseq	rand_net/50-60-10	96.33
2clseq	rand_net/60-40-10	16.03
2clseq	rand_net/60-60-10	180.45
2clseq	rand_net/60-60-5	172.22
2clseq	rand_net/70-40-10	35.02
2clseq	rand_net/70-40-5	52.15
2clseq	rand_net/70-60-10	188.11
2clseq	rand_net/70-60-5	611.71
2clseq	satex-c/c6288-s	0.73
2clseq	satex-c/c6288	0.77
berkmin	dinphil/dp10u09	321.67
berkmin	fpga-r/rcs_w9	2 054.50
berkmin	satex-c/cnf-r4-b2-k1.1	1 402.24
limmat	Homer/homer16	2 315.05
limmat	dinphil/dp12s12	5.21
modoc	Homer/homer15	843.87
zchaff	fifo8/fifo8_200	1 417.68
zchaff	w10/w10_70.cnf	661.74
zchaff	satex-/9vliw_bp_mc	1 274.94
zchaff	fvp-u.2.0/5pipe	186.58
zchaff	fvp-u.2.0/5pipe_3_oo	533.11
zchaff	fvp-u.2.0/5pipe_4_ooo	1 667.30
zchaff	fvp-u.2.0/5pipe_5_ooo	814.92
zchaff	fvp-u.2.0/7pipe_bug	452.14

Table 10 (Industrial) SOTA contributors for industrial instances are 2clseq, berkmin, limmat, modoc and zchaff. Note that 2clseq worked very well on rand_net benchmarks, same thing for zchaff and pipe instances.

Table 11 (Handmade) SOTA contributors for hand-made benchmarks are berkmin, limmat, rb2cl and zchaff.

Table 12 (Random) SOTA contributors for randomly generated benchmarks are 2clseq, dlmsat1, dlmsat3, oksolver, unitwalk, usat05, usat10.

In order to obtain a total order among the solvers, one must first classify benchmarks themselves. For this, the notion of SOTA contributors can be used [55]: benchmarks solved by all SOTA contributors are said *easy*, those solved by *at least one SOTA contributor* (but not all) are called *difficult*.[8] Note that the benchmarks not solved by any solver are not considered here.

Now, it is easy to rank the solvers accordingly to the number of *difficult* instances they can solve. Tables 13 and 14 provide a summary for SAT2002 competition, for respectively complete solvers and satisfiable benchmarks.

[8] A degree of difficulty can be computed using the ratio number of failing SOTA contributors over the total number of SOTA contributors.

Table 11
Uniquely solved Handmade benchmarks during the first stage.

Solver	Benchmark	CPU (s)
berkmin	hanoi6_on	295.04
berkmin	rope_1000	2 058.62
berkmin	x1.1_32	57.13
berkmin	x1_32	82.18
limmat	pyhala-braun-sat-40-4-01	1 295.77
rb2cl	lisa21_1_a	1 955.62
zchaff	lisa21_2_a	287.74
zchaff	pyhala-braun-sat-40-4-02	971.59
zchaff	pyhala-braun-unsat-35-4-01	1 474.30
zchaff	pyhala-braun-unsat-35-4-02	1 653.80
zchaff	Urquhart-s3-b5	1 507.11
zchaff	x1.1_36	786.60
zchaff	x2_36	1 828.81

What can we conclude? First of all, we awarded SOTA contributors. Looking at the SOTA ranking per category, berkmin, zchaff and oksolver would be awarded. Limmat, our fourth awarded, is most of the time third after zchaff and berkmin in industrial and hand-made categories. So the result of the SAT2002 competition looks reasonable. Berkmin certainly deserves a special attention from the community, since despite the bug that made it crashed on more than 100 benchmarks, it is ranked first in the industrial category using the SOTA system. This little impact of crashes can certainly be due to the fact that only the second SAT-engine of Berkmin crashed, which means that most instances on which Berkmin crashed are hard for all solvers anyway (this engine is called for hardest instances only).

Furthermore, we now have difficult and unsolved instances for the next competition. The degree of difficulty provided by the SOTA system can be used to tune solvers for the next competition: first try to solve instances of medium difficulty, then try the really hard ones. All that information will be included in the instances archive.

4.2. Graphical analysis of SOTA CPU performances

The SOTA ranking also allows to focus on subsets of solvers/benchmarks. We can for instance represent the above results in a graphical way, mixing this time complete and incomplete solvers. Doing this, we can take CPU time results into account to give a better picture of SOTA contributors performance. Figures 1–3 show for all respective SOTA contributors how much CPU time is needed to solve an increasing number of instances.

Such representation is important for the validation of the CPU time slice parameter. For this competition, it was arbitrarily chosen[9] and one can ask whether this value can play a crucial role in the results.

[9] Based only on the number of benchmarks/solvers and machines.

Table 12
Uniquely solved randomly generated benchmarks during the first stage.

Solver	Benchmark	CPU (s)
2clseq	5col100_15_1	1 148.08
2clseq	5col100_15_4	1 097.78
2clseq	5col100_15_5	1 018.13
2clseq	5col100_15_6	1 353.14
2clseq	5col100_15_7	748.21
2clseq	5col100_15_8	1 160.66
2clseq	5col100_15_9	821.42
dlmsat1	hgen5-v300-s1895562135	391.94
dlmsat1	hgen5-v300-s528975468	512.81
dlmsat1	hgen2-v300-s1807441418	651.35
dlmsat1	hgen3-v450-s646636548	141.16
dlmsat1	hgen4-v200-s2074278220	16.79
dlmsat1	hgen4-v200-s812807056	1 158.20
dlmsat3	hgen3-v450-s356974048	821.42
dlmsat3	4col280_9_2	1 766.60
dlmsat3	4col280_9_4	6.60
dlmsat3	4col280_9_6	1 792.59
dlmsat3	4col280_9_7	108.43
oksolver	glassy-v399-s1993893641	1 162.07
oksolver	glassy-v399-s524425695	814.44
oksolver	glassybp-v399-s1499943388	273.31
oksolver	glassybp-v399-s944837607	615.27
oksolver	glassyb-v399-s1267848873	1 001.71
oksolver	5cnf_3900_3900_160	243.24
oksolver	5cnf_3900_3900_170	911.32
oksolver	5cnf_3900_3900_180	1 790.46
oksolver	5cnf_3900_3900_190	1 778.96
oksolver	5cnf_4300_4300_090	1 390.55
oksolver	5cnf_4300_4300_100	1 311.92
unitwalk	hgen2-v650-s2139597266	536.09
unitwalk	hgen2-v700-s543738649	32.38
unitwalk	hgen2-v700-s548148704	0.62
unitwalk	hgen3-v500-s1754870155	157.03
usat05	okgen-c2550-v600-s552691850	5.57
usat10	hgen3-v450-s1400022686	0.39

The first observation from all the figures is that, in general, the order of the respective curves does not change after a certain CPU time (there is mostly no crossing of lines after 100 s). There are however two exceptions. First, on figure 1, for 2clseq and zchaff: 2clseq did not solve any other problem after 1000 s, which allowed zchaff to solve more problems in the given CPU time. 2clseq would probably have been considered in a better way if the cut-off had been fixed to less than 1000 s. Secondly, on Handmade benchmarks, the Berkmin curve crosses the one of zchaff just after 2000 s (this can be due to the internal bug of Berkmin). One can ask the question of what happened to the results if

Table 13
Number of difficult SAT+UNSAT benchmarks solved by each solver during the first stage.

Industrial		Handmade		Randomly generated	
Solver	# solved	Solver	# solved	Solver	# solved
berkmin	121	berkmin	68	oksolver	548
zchaff	104	zchaff	68	marchII	543
2clseq	93	limmat	47	marchIIse	543
limmat	87	simo	43	2clseq	369
simo	57	2clseq	34	rb2cl	338
rb2cl	36	oksolver	33	zchaff	305
oksolver	35	jquest	16	simo	298
jquest	31	marchIIse	13	berkmin	263
marchIIse	29	rb2cl	10	limmat	221
modoc	26	marchII	9	modoc	219
blindsat	20	modoc	7	jquest	122
marchII	16	blindsat	0	blindsat	3

Table 14
Number of difficult satisfiable benchmarks solved by each solver during the first stage.

Industrial		Handmade		Randomly generated	
Solver	# solved	Solver	# solved	Solver	# solved
berkmin	68	zchaff	35	oksolver	548
zchaff	64	berkmin	33	marchII	543
limmat	54	limmat	29	marchIIse	543
2clseq	46	simo	27	2clseq	369
unitwalk	41	oksolver	23	rb2cl	338
simo	39	2clseq	21	zchaff	305
dlmsat2	37	unitwalk	11	simo	298
dlmsat1	36	dlmsat1	11	berkmin	263
rb2cl	35	marchIIse	10	dlmsat1	255
dlmsat3	35	dlmsat3	10	dlmsat2	234
saturn	34	saturn	9	dlmsat3	233
usat10	33	usat05	9	unitwalk	227
usat05	31	usat10	9	limmat	221
oksolver	30	dlmsat2	9	modoc	219
marchIIse	26	marchII	7	saturn	208
jquest	24	rb2cl	6	usat10	206
blindsat	20	modoc	5	usat05	205
modoc	20	jquest	4	jquest	122
ga	18	blindsat	0	blindsat	3
marchII	13	ga	0	ga	3

the CPU time slice was less than 2000 s. At last, as expected, one can notice on figure 3 that uncomplete solvers obtain the best performances on SAT instances (the rightmost curve is oksolver on SAT+UNSAT benchmarks). Obviously – oksolver is complete – the picture is quite different as soon as we take into account all (SAT+UNSAT) answers

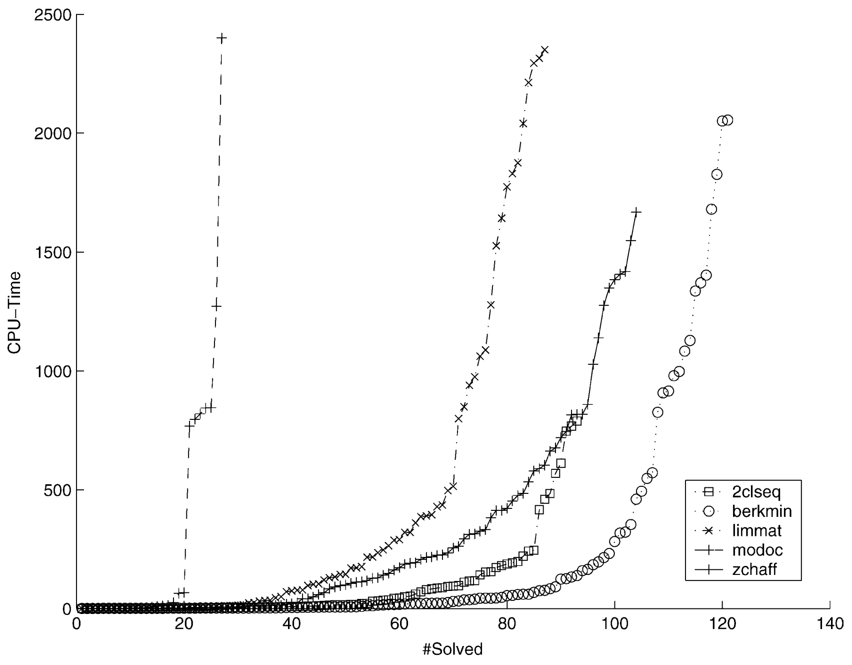

Figure 1. Number of instances solved vs. CPU time for SOTA contributors: Industrial benchmarks.

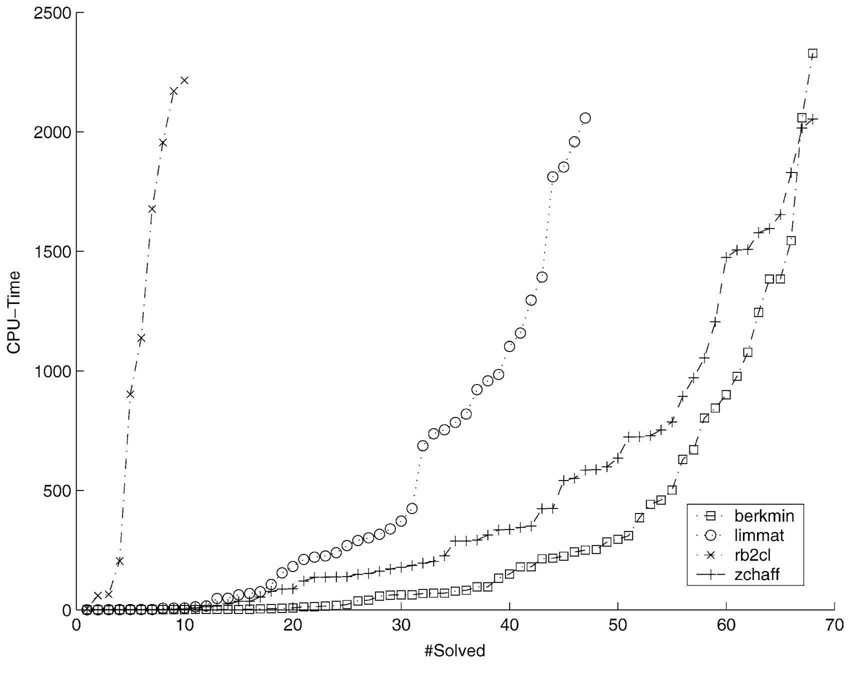

Figure 2. Number of instances solved vs. CPU time for SOTA contributors: Handmade benchmarks.

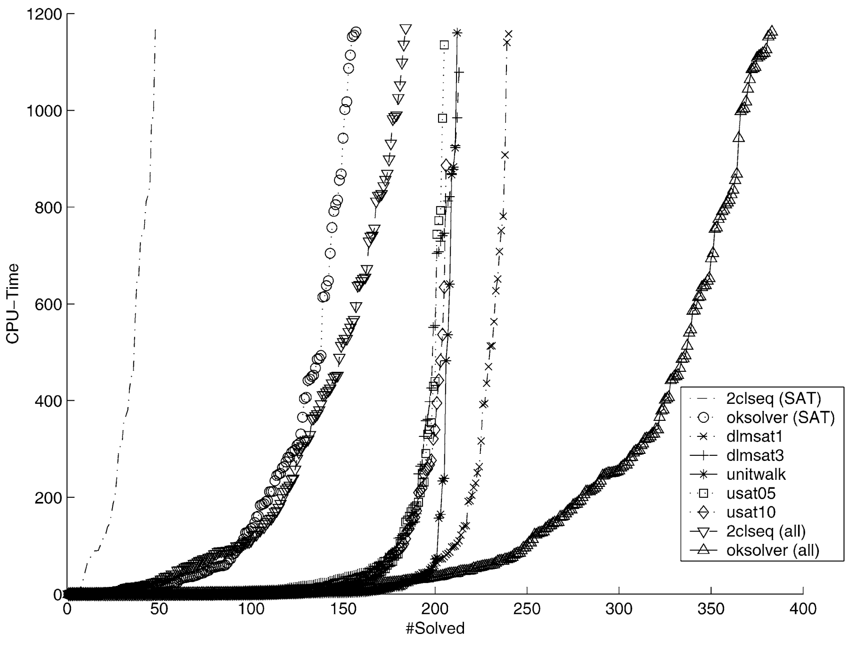

Figure 3. Number of instances solved vs. CPU time for SOTA contributors: Randomly generated benchmarks (note that the CPU time slice was smaller for these benchmarks). For each complete solvers (2clseq and oksolver), we plot two curves: (1) a curve for SAT results only (in order to compare their performances to uncomplete solvers) and (2) for all benchmarks (SAT+UNSAT).

for oksolver. It may be at last interesting to notice that all uncomplete solvers obtain more or less the same curve (observe how close are the curves for dlmsat1, dlmsat2, unitwalk, usat05 and usat10).

Two conclusions may be drawn. First, the growth of the curves on industrial benchmarks clearly shows that 2500 s is enough. Moreover, we can guess that a CPU time of 1500 s would have been sufficient. On hand-made instances, the picture is not so clear. May be this is due to the scalability of generated instances, which sometimes allows a smooth growth of the needed CPU time (a smooth growth may also be observed on random instances).

4.3. *Cumulative CPU analysis of results*

There is a lot of different ways to study the data collected during the first stage. Here, we give a ranking similar to the one of SAT-Ex. Each solver is given a maximal amount of time to solve an instance (recall that the launching heuristic also applies here). If it cannot solve the instance, then we only penalize it with the maximal CPU time. The ranking is given by the sum of all CPU time. However, this kind of ranking implies to penalize solvers that cannot solve a given instance, instead of just counting successes, we also have to count failures. So, a problem occurs with incomplete solvers on Unknown instances (whose can be Unsat). Thus, we reconsider how to compare

Table 15
Cumulative CPU time on Industrial benchmarks. Beware, we partition here complete vs. incomplete solvers.

Complete			Incomplete		
Solver	CPU (hours)	# solved (245)	Solver	CPU (hours)	# solved (137)
berkmin	54	175	**unitwalk**	54	57
zchaff	66	163	dlmsat1	57	53
2clseq	70	146	dlmsat2	57	54
limmat	79	144	saturn	60	49
simo	97	105	dlmsat3	63	51
rb2cl	114	81	usat05	64	44
oksolver	115	78	usat10	64	46
modoc	117	79	ga	79	20
marchIIse	118	72			
jquest	122	71			
marchII	128	59			
blindsat	146	25			

Table 16
Cumulative CPU time on Handmade benchmarks. Beware, we partition here complete vs. incomplete solvers.

Complete			Incomplete		
Solver	CPU (hours)	# solved (276)	Solver	CPU (hours)	# solved (156)
berkmin	83	160	**dlmsat1**	71	56
zchaff	88	160	dlmsat3	71	54
limmat	99	139	dlmsat2	76	52
simo	105	134	saturn	80	44
2clseq	107	125	unitwalk	84	45
oksolver	109	122	usat05	84	33
rb2cl	120	102	usat10	85	31
jquest	126	98	ga	96	15
marchII	129	91			
marchIIse	129	98			
modoc	130	89			
blindsat	172	18			

solvers, and we rather partition solvers in Complete/Uncomplete solvers rather than by solver/benchmarks (e.g., Complete on ALL, All on SAT).

Results are given in tables 15 (Industrial), 16 (Handmade) and 17 (Random). For randomized solvers, the median CPU time needed for a given benchmark is taken into account. As already noted using the SOTA system, berkmin is in head on industrial benchmarks, the whole picture tends to reinforce our awards, in all categories.

Table 17
Cumulative CPU time on Random benchmarks. Beware, we partition here complete vs. incomplete solvers.

Complete			Incomplete		
Solver	CPU (hours)	# solved (1502)	Solver	CPU (hours)	# solved (1502)
oksolver	248	834	**dlmsat1**	325	541
marchII	253	829	unitwalk	332	513
marchIIse	254	829	dlmsat2	332	517
2clseq	310	655	dlmsat3	335	519
rb2cl	316	616	usat10	339	492
zchaff	326	573	usat05	340	491
berkmin	333	541	saturn	345	493
simo	337	569	ga	462	116
limmat	361	461			
modoc	370	448			
jquest	422	252			
blindsat	464	115			

5. Difficulties and future competitions

Despite a good maturity level in solvers, pure-SAT competitions are not so frequent (the previous one took place 7 years ago). In some aspect, this competition has surpassed all previous competitions (in the number of benchmarks and solvers, the availability of results on the web), but some choices were hard to take. Some of them were good, some were not. Because another competition will be held next year, it is important to take stock of this one, in order to think about the next one.

5.1. Some lessons

Let us begin our first assessments with our own difficulties during the competition. Despite of our efforts to make everything automatic, life is always more complex than predictions. In particular, we got unexpected bugs (or should we say that we did not expect too many *expected* unexpected bugs?), in all the stages of the competition, including bugs in benchmarks:

– problems with input/output format (solvers),
– duplicate literals or opposite literals in clauses (benchmarks),
– internal timeout hardcoded in solvers,
– wrong declaration of benchmarks (SAT vs. UNSAT) and solvers (randomized vs. deterministic);

Even worse, the side effects of bugs were often important (e.g., if an instance is buggy, it should be dropped, which affects the whole procedure; the running scripts had to be modified to ignore buggy results). Thus, some experiments had to be repeated which had lead to smaller amount of available CPU time than we expected. For all these reasons, human action was frequently needed to understand whether a bug was due to solver, due to benchmark, or due to running scripts, and, of course, how to fix it appropriately.

Despite we tried to set up as strict rules as possible, still human action was needed in the choice of benchmarks (how to separate them into series/categories; how many benchmarks to take for the second stage?; how many benchmarks to generate from each random generator?). The human action was especially hard to implement since one of the organizers submitted his own solver and benchmarks, another one submitted benchmarks (though not qualified for the awards), and the remaining one planned to submit a solver (but did not). Therefore, Laurent Simon had to make a lot of decisions almost alone while was extremely busy with actually running the competition.

Also a wrong decision has been made concerning the selection of benchmarks *from* series. We expected that larger benchmarks of the same series should be harder, and therefore decided not to run a solver on larger benchmarks if it failed on smaller benchmarks of the same series. However, our conjecture was false (especially for industrial problems, where BMC SAT-checking of depth n can be harder than depth $n + 1$, and for some of the random generators, where the conjecture can be true for the median CPU time over a lot of formulas, but false for just 4 formulas), and it produced completely wrong decisions concerning mixed SAT/UNSAT series (clearly, any incomplete solver fails all (smaller) unsatisfiable benchmarks and thus is not run on (larger) satisfiable ones). We had to fix the effect of this either by dropping SAT/UNSAT series from the consideration for incomplete solvers, or by running the experiments on all benchmarks (possibly wasting CPU time).

Two other difficulties we encountered were (briefly):

- SAT-Ex scripts had to be tuned during the competition, e.g., to present results according to the competition rules, and to process solvers and benchmarks differently in different categories;
- the lack of live action (the system was not automated enough to put everything on the web without the help of the organizers who were extremely busy; also the results would be quite misleading if putted on the web immediately because once a bug had been eventually found, it changed the picture of the competition).

However, despite of the problems emphasized above, we treat the whole competition as a success in many aspects.

5.2. About the next competition

Because any of the SOTA contributors is important for SAT research, we are thinking about delivering a SOTA contributor certificate for the next competition. But, to adopt the SOTA system for awarding solvers, we need a better classification of available SAT benchmarks as in TPTP. Let us emphasize some differences between the CASC competition (where SOTA ranking *is* the rule) and the SAT2002 competition. Differences are essentially due to the different level of maturity of these competitions:

1. In the CASC competition, most instances are part of the TPTP library, so they are well known and classified. This is not the case for the SAT competition since we received a lot of completely new benchmarks for running the competition.
2. The ranking proposed above is made on *Specialist Problem Classes*, whose granularity is finer than our simple (Industrial, Handmade, Randomly generated) partition (for instance, the pigeon-hole problem may occur as an industrial problem … How can we classify such a benchmark?).
3. We used a heuristic to save CPU time, so not all systems were run on all instances.

Many other improvements are possible for the next competition. Still it seems like human action is unavoidable. To make it more fair and less time-consuming for the organizers, we propose to appoint a board of *judges* similarly to CASC. When the competition rules do not give an explicit answer, it is up to the judges to decide what to do. They can also play a role for instance in the partition of benchmarks.

For the **future competitions** we propose the following:

– Allow more time for submitters to experiment with their solvers (prior to submission) on one of the *actual* computers of the competition.
– Make clear which version of a solver is used. Sometimes, under the same name, quite different algorithms or techniques are used (e.g., Berkmin 56 that is a single engine solver whose results were published in [21] and Berkmin 62 that was the 2 engines solver submitted to the competition). This would prevent a spectator from a suspicion when an instance is solved by a particular solver as reported in a previously published paper but not during the competition.
– Make competition more automatic both technically (better scripts) and semantically (more strict rules with no exceptions). This could allow to make the results online as soon as they are available, for instance to allow solver submitters to check the behavior of their solver during the first stage.
– After benchmarks are selected, every solver is run on every benchmark even if it seems a waste of CPU time (however, solver submitters could be allowed to submit their solvers only for some categories of benchmarks).
– Run the winners of the previous competition anyway (or the SOTA contributors). If an old winner wins a category, no award is given for this category. The same principle applies for the 2nd stage benchmarks.
– Limit the number of submissions per author groups: if one technique performs well in a category, it is likely that all its variants perform equally well (see, e.g., dlmsat1/2/3 in 1st stage random SAT category). Then the second stage is biased. One solution is to limit the number of variants and to qualify only the best one for the final stage.
– Provide right now the scripts/programs used to check input/output/instances format to delegate that step to the submitters. (Then a fully automatic machinery can reject submissions not respecting the formats.)

Some other points, more or less prospective, are possible:

– Classify new benchmarks accordingly to the performances of previous year SOTA contributors. For instance, "*easy*" benchmarks (according to SOTA) could thus be discarded, freeing CPU time.
– Discuss the notion of series and how to score solvers on series. For instance, if many benchmarks in a series are easy, then most of solvers would have solved the series. This can be resolved for instance by giving the solvers points according to their performance for each particular series (and not 1 vs. 0 as it was in SAT 2002 competition). For example, the series winner could get 5 points, the next solver could get 4, etc.
– More prospective, one can use only a timeout per series instead of a timeout per instance (maybe as a new category). A solver able to solve quickly the first instances of a series will have more time to solve the remaining instances. Beware here: the order in which the instances are provided to the solver matters, and it is not easy to determine the best order.
– At last, we could use a larger cluster of machines, as the ones that they use for VLSI/CAD applications and simulations, where very large empirical evaluations are often performed. For instance, the new version of `bookshelf.exe` [9] should allow to use hundreds of identical computers with a simple-web interface (automatic report, ...). However, modifications are needed to merge the SAT-Ex architecture into this cluster interface.

The final point is to run the next competition only before the SAT Symposium (only publishing results and giving awards at the meeting) ...

6. Conclusions

The competition revealed many expected results as well as a few surprising ones. First of all, *incomplete solvers* appeared to be much weaker than complete ones. Note that no incomplete solver won any category of satisfiable formulas while these categories were intended rather for them than for complete solvers (note that in theory randomized one-sided error algorithms accept potentially more languages than deterministic ones). In Industrial and Handmade categories only one incomplete solver (UnitWalk) was in the top five. This can be due to the need of specific tuning for local search algorithms (noise, length of walk, etc.) and, probably, to the lack of new ideas for the use of randomization (except for local search). If automatic tuning (for similar benchmarks) of incomplete solvers is possible, how to incorporate it in the competition?

On Industrial and Handmade benchmarks, *zchaff and algorithms close to it* (Berkmin and limmat) were dominating. On the other hand, the situation on *randomly generated instances* was quite different. These algorithms were not even in the top five! Also, randomly generated satisfiable instances were the only category where incomplete algorithms were competitive (four of them: dlmsat1(2,3) and UnitWalk, were in the top five).

Concerning the unsatisfiable instances, the top five list is also looking quite differently to other categories.

In fact, only two solvers appeared in all the top five lists for the three categories Industrial/Handmade/Random: a *non*-zchaff-like complete solver 2clseq for SAT+UNSAT and an incomplete solver UnitWalk for SAT. However, they did not win anything. The (unsurprising) conclusion is that specialized solvers indeed perform better on the classes of benchmarks they are specialized for. Also it confirms that our choice of categories was right. But maybe an award should be given to algorithms performing uniformly on all kinds of instances (while some part of the community was against an "overall winner" for the present competition).

Another conclusion of the competition is that almost all benchmarks that remained *unsolved within 40 minutes* on P-III-450 (or a close number of CPU cycles on a faster machine) have not been solved in 6 hours either. This can be partially due to the fact that few people experimented with the behaviour of their solvers for that long. Note that the greatest number of second stage benchmarks was solved in Industrial–SAT+UNSAT category, the one where probably the greatest number of experiments is made by the solvers authors. Also many solvers crashed on huge formulas (probably due to the lack of memory).

It is no surprise that the *smallest unsolved unsatisfiable* benchmark (xor-chain instance by L. Zhang) belongs to Handmade category. In fact, many of unsatisfiable benchmarks in this category are also very small. However, it seems like all these benchmarks are hard only for resolution (and hence DP- and DLL-like algorithms) where exponential lower bounds are known for decades (see, e.g., [56,57]). Therefore, if non-resolution-based complete algorithms come, these benchmarks will be probably easy for them. For example, LSAT (fixed version) and eqsatz (not participated) which employ equality reasoning can easily solve parity32 instances that remained unsolved in the competition.

On the other hand, the *smallest unsolved* (provably) *satisfiable* benchmark (hgen2 instance by Edward A. Hirsch) is made using a random generator. Other small hard satisfiable benchmarks also belong to Random category. These benchmarks are much larger than hard unsatisfiable ones (5250 vs. 844 literal occurrences). This is partially due to the fact that no exponential lower bounds are known for DPLL-like algorithms for *satisfiable* formulas (in fact, the only such lower bounds we know for other SAT algorithms are of [23]). In contrast, no random generator was submitted for (provably) unsatisfiable instances. (Of course, some of the handmade unsatisfiable instances can be generated using random structures behind them; however, this does not give a language not known to be in coRP (and even in ZPP).) Note that the existence of an efficient generator of a *coNP-complete* language would imply NP $=$ coNP (random bits form a short certificate of membership).

Probably, at the end, the main thing about competition is that it attracted completely new solvers (e.g., 2clseq and limmat) and a lot of new benchmarks.

Some challenging questions, drawn from the conclusion, are:

1. Construct a *random* generator for a "hard" language of provably *unsatisfiable* formulas.

2. Design an incomplete algorithm that would outperform complete algorithms (on satisfiable formulas), or explain why it is not possible.
3. Construct *satisfiable* formulas giving exponential lower bounds for the worst-case running time of *DPLL-like* algorithms.[10]
4. For the competition: how to request/represent a certificate for "unsatisfiable" answers without violating the rights of non-DPLL-like algorithms?

Acknowledgements

We would like to thank John Franco, Michal Kouril, and the University of Cincinnati for providing us the computers and local maintenance. We also thank Hans van Maaren and the computational logic group of the Technical University of Delft for providing the awards. Of course, not to forget the many benchmarks and solvers authors. We are also very grateful to all SAT2002 conference participants for their feedback.

References

[1] P.A. Abdulla, P. Bjesse and N. Eén, Symbolic reachability analysis based on SAT-solvers, in: *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)* (2000).

[2] F. Bacchus, Enhancing Davis Putnam with extended binary clause reasoning, in: *Proceedings of National Conference on Artificial Intelligence (AAAI-2002)* (2002).

[3] F. Bacchus, Exploring the computational tradeoff of more reasoning and less searching, in: [49, pp. 7–16] (2002).

[4] L. Baptista and J.P. Marques-Silva, Using randomization and learning to solve hard real-world instances of satisfiability, in: *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP)* (2000).

[5] R.J.J. Bayardo and R.C. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)* (AMS, Providence, RI, 1997) pp. 203–208.

[6] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita and Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in: *Proceedings of Design Automation Conference (DAC'99)* (1999).

[7] M. Buro and H.K. Büning, Report on a SAT competition, Bulletin of the European Association for Theoretical Computer Science 49 (1993) 143–151.

[8] C.-M. Li, B. Jurkowiak and P.W. Purdom Jr, Integrating symmetry breaking into a DLL procedure, in: [49, pp. 149–155] (2002).

[9] A.E. Caldwell, A.B. Kahng and I.L. Markov, Toward CAD-IP reuse: The MARCO GSRC bookshelf of fundamental CAD algorithms, IEEE Design and Test (May 2002) 72–81.

[10] P. Chatalic and L. Simon, Multi-resolution on compressed sets of clauses, in: *Twelth International Conference on Tools with Artificial Intelligence (ICTAI'00)* (2000) pp. 2–10.

[11] S.A. Cook, The complexity of theorem-proving procedures, in: *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science* (1971) pp. 151–158.

[10] The question has been partially resolved in M. Alekhnovich, E.A. Hirsch, D. Itsykson, Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas, in: *Proceedings of ICALP 2004*. Springer, to appear.

[12] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella and M. Vardi, Benefits of bounded model checking at an industrial setting, in: *Proc. of CAV* (2001).

[13] E. Dantsin, A. Goerdt, E.A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan and U. Schöning, Deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search, Theoretical Computer Science 189(1) (2002) 69–83.

[14] M. Davis, G. Logemann and D. Loveland, A machine program for theorem proving, Communications of the ACM 5(7) (1962) 394–397.

[15] M. Davis and H. Putnam, A computing procedure for quantification theory, Journal of the ACM 7(3) (1960) 201–215.

[16] O. Dubois, P. André, Y. Boufkhad and J. Carlier, SAT versus UNSAT, in: [29, pp. 415–436] (1996).

[17] O. Dubois and G. Dequen, A backbone-search heuristic for efficient solving of hard 3-SAT formulae, in: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, Seattle, WA (2001).

[18] M.D. Ernst, T.D. Millstein and D.S. Weld, Automatic SAT-compilation of planning problems, in: [28, pp. 1169–1176] (1997).

[19] F. Aloul, A. Ramani, I. Markov and K. Sakallah, Solving difficult SAT instances in the presence of symmetry, in: *Design Automation Conference (DAC)*, New Orleans, LO (2002) pp. 731–736.

[20] J.W. Freeman, Improvements to propositional satisfiability search algorithms, Ph.D. thesis, Departement of Computer and Information Science, University of Pennsylvania, Philadelphia, PA (1995).

[21] E. Goldberg and Y. Novikov, BerkMin: A fast and robust SAT-solver, in: *Design, Automation, and Test in Europe (DATE '02)* (2002) pp. 142–149.

[22] C.P. Gomes, B. Selman and H. Kautz, Boosting combinatorial search through randomization, in: *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, Madison, WI (1998) pp. 431–437.

[23] E.A. Hirsch, SAT local search algorithms: Worst-case study, Journal of Automated Reasoning 24(1/2) (2000) 127–143. Also reprinted in *Highlights of Satisfiability Research in the Year 2000*, Frontiers in Artificial Intelligence and Applications, Vol. 63 (IOS Press, 2000).

[24] E.A. Hirsch, New worst-case upper bounds for SAT, Journal of Automated Reasoning 24(4) (2000) 397–420. Also reprinted in *Highlights of Satisfiability Research in the Year 2000*, Frontiers in Artificial Intelligence and Applications, Vol. 63 (IOS Press, 2000).

[25] E.A. Hirsch and A. Kojevnikov, UnitWalk: A new SAT solver that uses local search guided by unit clause elimination, Annals of Mathematics and Artificial Intelligence 43 (2005) 91–111.

[26] J.N. Hooker, Needed: An empirical science of algorithms, Operations Research 42(2) (1994) 201–212.

[27] J.N. Hooker, Testing heuristics: We have it all wrong, Journal of Heuristics (1996) 32–42.

[28] IJCAI97, *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, Nagoya, Japan (1997).

[29] D. Johnson and M. Trick (eds.), *Second DIMACS Implementation Challenge: Cliques, Coloring and Satisfiability*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26 (American Mathematical Society, 1996).

[30] H. Kautz and B. Selman (eds.), *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)* (Elsevier Science, 2001).

[31] H.A. Kautz and B. Selman, Planning as satisfiability, in: *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI'92)* (1992) pp. 359–363.

[32] H.A. Kautz and B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, in: *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'96)* (1996) pp. 1194–1201.

[33] E. Koutsoupias and C.H. Papadimitriou, On the greedy algorithm for satisfiability, Information Processing Letters 43(1) (1992) 53–55.

[34] O. Kullmann, First report on an adaptive density based branching rule for DLL-like SAT solvers, using a database for mixed random conjunctive normal forms created using the Advanced Encryption Standard (AES), Technical Report CSR 19-2002, University of Wales Swansea, Computer Science Report Series (2002). (Extended version of [36].)

[35] O. Kullmann, Investigating the behaviour of a SAT solver on random formulas, Annals of Mathematics and Artificial Intelligence (2002).

[36] O. Kullmann, Towards an adaptive density based branching rule for SAT solvers, using a database for mixed random conjunctive normal forms built upon the Advanced Encryption Standard (AES), in: [49] (2002).

[37] C.-M. Li, A constrained based approach to narrow search trees for satisfiability, Information Processing Letters 71 (1999) 75–80.

[38] C.-M. Li, Integrating equivalency reasoning into Davis–Putnam procedure, in: *Proceedings of the 17th National Conference in Artificial Intelligence (AAAI'00)*, Austin, TX (2000) pp. 291–296.

[39] C.-M. Li and Anbulagan, Heuristics based on unit propagation for satisfiability problems, in: [28, pp. 366–371] (1997).

[40] I. Lynce and J.P. Marques Silva, Efficient data structures for backtrack search SAT solvers, in: [49] (2002).

[41] I. Lynce, L. Baptista and J.P. Marques Silva, Stochastic systematic search algorithms for satisfiability, in: [30] (2001).

[42] J.P. Marques-Silva and K.A. Sakallah, GRASP – A new search algorithm for satisfiability, in: *Proceedings of IEEE/ACM International Conference on Computer-Aided Design* (1996) pp. 220–227.

[43] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang and S. Malik, Chaff: Engineering an efficient SAT solver, in: *Proceedings of the 38th Design Automation Conference (DAC'01)* (2001) pp. 530–535.

[44] F. Okushi and A. Van Gelder, Persistent and quasi-persistent lemmas in propositional model elimination, in: *(Electronic) Proc. 6th Int'l Symposium on Artificial Intelligence and Mathematics* (2000).; Annals of Mathematics and Artificial Intelligence 40(3–4) (2004) 373–402.

[45] R. Ostrowski, E. Grégoire, B. Mazure and L. Sais, Recovering and exploiting structural knowledge from CNF formulas, in: *Proc. of the Eighth International Conference on Principles and Practice of Constraint Programming (CP'2002)*, Ithaca, NY (2002).

[46] R. Paturi, P. Pudlák and F. Zane, Satisfiability coding lemma, in: *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS'97* (1997) pp. 566–574.

[47] S. Prestwich, A SAT approach to query optimization in mediator systems, in: [49, pp. 252–259] (2002).

[48] S.D. Prestwich, Randomised backtracking for linear pseudo-Boolean constraint problems, in: *Proceedings of Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems* (2002).

[49] SAT2002, *Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, Cincinnati, OH (2002).

[50] R. Schuler, U. Schöning, O. Watanabe and T. Hofmeister, A probabilistic 3-SAT algorithm further improved, in: *Proceedings of 19th International Symposium on Theoretical Aspects of Computer Science, STACS 2002* (2002).

[51] B. Selman, H.A. Kautz and B. Cohen, Noise strategies for improving local search, in: *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI'94)*, Seattle (1994) pp. 337–343.

[52] B. Selman, H. Levesque and D. Mitchell, A new method for solving hard satisfiability problems, in: *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)* (1992) pp. 440–446.

[53] Y. Shang and B.W. Wah, A discrete Lagrangian-based global-search method for solving satisfiability problems, Journal of Global Optimization 12(1) (1998) 61–99.

[54] L. Simon and P. Chatalic, SATEx: a Web-based framework for SAT experimentation, in: [30] (2001); `http://www.lri.fr/~simon/satex`.

[55] G. Sutcliff and C. Suttner, Evaluating general purpose automated theorem proving systems, Artificial Intelligence 131 (2001) 39–54.

[56] G.S. Tseitin, On the complexity of derivation in the propositional calculus, in: *Structures in Constructive Mathematics and Mathematical Logic, Part II*, ed. A.O. Slisenko (Consultants Bureau, New York, 1970) pp. 115–125. Translated from Russian.

[57] A. Urquhart, Hard examples for resolution, Journal of the Association for Computing Machinery 34(1) (1987) 209–219.

[58] A. Van Gelder, Autarky pruning in propositional model elimination reduces failure redundancy, Journal of Automated Reasoning 23(2) (1999) 137–193.

[59] A. Van Gelder, Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution, in: *Seventh Int'l Symposium on AI and Mathematics*, Fort Lauderdale, FL (2002).

[60] A. Van Gelder, Generalizations of watched literals for backtracking search, in: *Seventh Int'l Symposium on AI and Mathematics*, Fort Lauderdale, FL (2002).

[61] A. Van Gelder and F. Okushi, Lemma and Cut strategies for propositional model elimination, Annals of Mathematics and Artificial Intelligence 26(1–4) (1999) 113–132.

[62] A. Van Gelder and Y.K. Tsuji, Satisfiability testing with more reasoning and less guessing, in: [29, pp. 559–586] (1996).

[63] M. Velev and R. Bryant, Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors, in: *Proceedings of the 38th Design Automation Conference (DAC '01)* (2001) pp. 226–231.

[64] J. Warners and H. van Maaren, Solving satisfiability problems using elliptic approximations: Effective branching rules, Discrete Applied Mathematics 107 (2000) 241–259.

[65] H. Zhang, SATO: An efficient propositional prover, in: *Proceedings of the International Conference on Automated Deduction (CADE'97)*, Lecture Notes in Artificial Intelligence, Vol. 1249 (1997) pp. 272–275.

[66] H. Zhang and M.E. Stickel, An efficient algorithm for unit propagation, in: *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale, FL (1996).

[67] L. Zhang, C.F. Madigan, M.W. Moskewicz and S. Malik, Efficient conflict driven learning in a Boolean satisfiability solver, in: *International Conference on Computer-Aided Design (ICCAD'01)* (2001) pp. 279–285.

[68] L. Zheng and P.J. Stuckey, Improving SAT using 2SAT, in: *Proceedings of the Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, ed. M.J. Oudshoorn, Melbourne, Australia (2002).