Evaluating *SAT on TANCS 2000 benchmarks

Armando Tacchella

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
tac@dist.unige.it

Abstract. In this paper we evaluate *SAT performance on TANCS 2000 benchmarks for the modal logic K. After a brief overview of *SAT implementation, we detail which *SAT options are more relevant in the context of our analysis. Then we present the experimental results: collected data include the CPU time, the number of consistency checks, and the allocated memory, to give a complete picture of *SAT behaviour on the selected benchmarks.

1 Algorithms and implementation

*SAT is a platform for the development of decision procedures for modal logics. Currently, *SAT features decision procedures for the normal modal logic K(m) and for the classical modal logic E(m) (see [1] for a presentation of these logics). In the spirit of [2], *SAT core algorithm can be described by two mutually recursive procedures:

- a SAT checker (see [3]) which *generates* assignments that satisfy the input formula, and
- a routine that *tests* the consistency of each assignment by checking the satisfiability of simpler modal formulas.

For lack of space, we do not elaborate further on *SAT algorithms: more details can be found in [4].

*SAT is implemented in C and should run flawlessly on most Unix platforms equipped with GNU development tools (`gcc` compiler, `make` and `m4` utilities). Libraries, code, and tools needed by *SAT are either publicly available or distributed with the code itself. For an in-depth discussion of *SAT modules, architecture and implementation please refer to [5]. *SAT distribution package, as well as most of the available relevant literature for this paper, can be found at:

`http://www.mrg.dist.unige.it/~tac/StarSAT.html`

2 Running *SAT

*SAT comes with many configurable options that the user selects either at compile-time (using compiler directives) or at run-time (using command line

Test (reference)	S/U/T	Cpu	Cons	Mem
C20-V2-D2	10/54/0	0.06	32	638
C20-V2-D2	0/64/0	0.16	157	896
C45-V8-D4	31/32/1	83.42	44418	3815
C45-V8-D5	10/28/26	186.56	88346	4689
C55-V8-D6	0/23/41	172.58	66443	6290

Fig. 1. Reference problems

Test (cnf)	S/U/T	Cpu	Cons	Mem
C10-V4-D4	4/0/0	102.05	26663	2720
C20-V4-D4	3/1/0	321.15	75967	2792
C30-V4-D4	0/4/0	18.08	3953	2802
C10-V8-D4	0/0/4	–	109023	–
C20-V8-D4	0/0/4	–	66377	–
C30-V8-D4	0/0/4	–	55121	–
C10-V16-D4	0/0/4	–	27740	–
C10-V4-D6	1/0/3	981.05	154892	4548
C20-V4-D6	0/0/4	–	152006	–
C10-V8-D6	0/0/4	–	56106	–
C20-V8-D6	0/0/4	–	39312	–

Test (modK)	S/U/T	Cpu	Cons	Mem
C10-V4-D4	0/4/0	30.20	20398	3414
C20-V4-D4	0/4/0	28.99	12796	3515
C30-V4-D4	0/4/0	48.49	28928	3745
C10-V8-D4	0/3/1	704.37	128334	9137
C20-V8-D4	0/0/4	–	262289	–
C30-V8-D4	0/1/3	727.93	96175	13017
C10-V4-D6	0/3/1	64.98	17100	5735
C20-V4-D6	0/1/3	295.58	213999	6388

Fig. 2. Benchmarks using optimized Halpern translation (`hard`)

switches). Most of these options are thoroughly described in *SAT manual [6], to which we point for further details. The results presented in this paper are obtained running *SAT with its default configuration, except for the timeout value that was increased to 1200 seconds of CPU time instead of 1000. Since the timeout includes the time required to build the internal data structures, and the samples of some problems are pretty big, our choice for the timeout value prevents preprocessing overhead to reduce raw search time.

Besides the very basic algorithm and data structures, *SAT currently features the following defaults:

- an optimized internal CNF conversion based on renaming ([7]), and a propositional search control strategy that efficiently takes care of added propositional labels;
- a modal lookahead optimization (called "early pruning" in [2]) that prevents modal inconsistencies from hiding in the propositional search;
- a combination of a caching structure and retrieval algorithms that provides a good balance among fast access time, small memory allocation and global speedup. *SAT caching mechanisms are described in greater detail in [8].

Test (cnfSSS)	S/U/T	Cpu	Cons	Mem
C10-V4-D4	8/0/0	0.08	246	1156
C20-V4-D4	6/2/0	0.48	945	1362
C30-V4-D4	2/6/0	0.66	1036	1434
C40-V4-D4	1/7/0	0.96	1268	1492
C50-V4-D4	0/8/0	0.66	726	1683
C10-V8-D4	8/0/0	0.25	643	1784
C20-V8-D4	8/0/0	3.28	4037	2584
C30-V8-D4	8/0/0	42.72	30507	3409
C40-V8-D4	2/6/0	58.20	32698	3736
C50-V8-D4	0/8/0	75.71	37776	4551
C10-V16-D4	8/0/0	0.46	906	3334
C20-V16-D4	8/0/0	6.57	6684	4565
C30-V16-D4	8/0/0	83.77	43580	5942
C40-V16-D4	2/0/6	604.25	234336	8195
C50-V16-D4	0/0/8	–	340587	–
C10-V4-D6	8/0/0	0.16	410	1455
C20-V4-D6	8/0/0	1.78	2437	1796
C30-V4-D6	7/1/0	7.46	7435	2088
C40-V4-D6	2/6/0	5.83	5396	2492
C50-V4-D6	0/8/0	4.66	3755	2502
C10-V8-D6	8/0/0	0.29	679	2346
C20-V8-D6	8/0/0	4.81	4808	3488
C30-V8-D6	8/0/0	90.65	52982	4262
C40-V8-D6	4/1/3	396.80	163496	5776
C50-V8-D6	0/0/8	–	393254	–
C10-V16-D6	8/0/0	0.73	1316	4370
C20-V16-D6	8/0/0	8.93	6701	5847
C30-V16-D6	8/0/0	176.68	72254	7918
C40-V16-D6	4/0/4	561.58	168941	10445
C50-V16-D6	0/0/8	–	248497	–

Test (modKSSS)	S/U/T	Cpu	Cons	Mem
C10-V4-D4	8/0/0	0.36	750	1501
C20-V4-D4	4/4/0	1.64	2231	1736
C30-V4-D4	3/5/0	2.83	2871	1830
C40-V4-D4	1/7/0	4.43	3461	1879
C50-V4-D4	0/8/0	4.41	3163	2451
C10-V8-D4	8/0/0	0.77	1302	2219
C20-V8-D4	8/0/0	12.77	10773	3230
C30-V8-D4	8/0/0	92.74	49964	4035
C40-V8-D4	2/5/1	517.84	203594	4384
C50-V8-D4	0/3/5	197.68	71422	5582
C10-V16-D4	8/0/0	1.29	1840	3850
C20-V16-D4	8/0/0	75.98	45052	5095
C30-V16-D4	6/0/2	703.53	244983	7068
C40-V16-D4	0/0/8	–	307407	–
C50-V16-D4	0/0/8	–	261662	–
C10-V4-D6	8/0/0	0.50	973	1691
C20-V4-D6	8/0/0	4.86	5279	2102
C30-V4-D6	5/3/0	22.85	16895	2449
C40-V4-D6	1/7/0	33.29	18658	3307
C50-V4-D6	0/8/0	46.07	21509	3353
C10-V8-D6	8/0/0	0.78	1258	2666
C20-V8-D6	8/0/0	22.59	16482	4139
C30-V8-D6	5/0/3	681.74	296324	5988
C40-V8-D6	0/0/8	–	380903	–
C50-V8-D6	0/0/8	–	309856	–
C10-V16-D6	8/0/0	1.75	2064	4850
C20-V16-D6	8/0/0	54.70	25887	6679
C30-V16-D6	5/0/3	502.92	152995	9720
C40-V16-D6	0/0/8	–	253395	–
C50-V16-D6	0/0/8	–	180169	–

Fig. 3. Benchmarks using Schmidt-Schauss-Smolka translation (`easy`)

3 Experiments

Here we present the evaluation of *SAT decision procedure for the modal logic K.[1] We run TANCS 2000 benchmarks, category `MODAL PSPACE Division`, subcategory `Problems for Modal QBF`. The three subclasses `hard`, `easy`, and `medium/hard` are considered, and for each one we run the benchmark problems that can be found on the TANCS 2000 web site, or a subset thereof. For every problem, we also consider the "modalized" version.

We arranged our results in seven tables showed in Figure 1 through Figure 4. In particular:

– the table in Figure 1 collects the results on various reference problems. The top row shows the results that we obtain on 64 samples for the problem `C10-V2-D2` in the `hard` category while the center row shows the result that we obtain on 64 "modalized" samples of the same problem. The entries at the bottom collect the results about three benchmarks in the `easy` category that are suggested as additional reference by the organizers.

[1] All the tests run on a `Pentium II 350 Mhz` equipped with `256 Mb` of main memory and `128 Mb` of swap space, operating system `Linux Suse 6.2` (kernel version `2.2.10`) and compiler `gcc 2.91.66`.

– Figure 2 (left) collects the results that we obtain on 11 benchmarks problems in the `hard` class. For a subset of such problems (8 in number), the table in Figure 2 (right) shows the result that we obtain on the "modalized" versions. In both variants, the number of samples per problem is 4.
– Figure 3 collects the results that we obtain on 30 benchmark problems in the `easy` class. The table on the left shows the results for the "plain" benchmarks, while the table on the right shows the results for the "modalized" variants. The number of samples per problem is 8 for both variants.
– Figure 4 (left) collects the results that we obtain on 19 benchmarks problems in the `medium/hard` class, while Figure 4 (right) shows the results that we obtained on 23 "modalized" versions. For both variants, the number of samples per problem is 8.

Each table consists of five columns. The first column, named **Test**, lists the names of the benchmark problems in each selection. Problems are identified using the official TANCS2000 generator syntax: we write Cx-Vy-Dz to denote a modal formula translated from a QBF having x clauses, alternation depth z and y variables, at most, per each alternation. Each formula has a fixed number of 4 literals per clause. The other columns show the following data:

S/U/T is the number of (S)atisfiable/(U)nsatisfiable/(T)imeout samples according to *SAT; we do not provide a column for memory outs since none of the runs exceeds the physical memory available on our machine;

Cpu is the average running time of *SAT in seconds, including preprocessing and internal CNF conversions; a dash in this column appears when all the samples of the problem exceed the time limit;

Cons is the average number of consistency checks, i.e. the average number of times that *SAT opens up a new modal branch in the search;

Mem is the maximum dynamic memory (in Kbytes) allocated by *SAT down any search path; a dash in this column appears when all the samples of the problem exceed the time limit.

Notice that "average number" here, has a different meaning depending on whether *SAT ends up in a timeout condition or not. If there is at least one sample on which *SAT does not exceed the time limit, the values for **Cpu**, **Cons** and **Mem** are the geometric average on *all* the samples that *do not exceed the time limit*. We decided this setting because, when a sample ends up in a timeout condition, **Cpu** and **Mem** measurements are either not significant (**Cpu** could be way bigger than the time limit) or not reliable (this happens for **Mem** in our implementation of timeouts). If *SAT exceeds the lime limit on all the samples, **Cpu** and **Mem** values are void. A different policy is used for **Cons** values: when all the samples exceed the time limit, **Cons** value is the geometric average on all the samples. In this case, **Cons** denotes the number of consistency checks performed, on the average, up to the time limit.

References

1. B. F. Chellas. *Modal Logic – an Introduction.* Cambridge University Press, 1980.

Test (cnfLadn)	S/U/T	Cpu	Cons	Mem
C10-V4-D4	8/0/0	404.29	314270	2546
C20-V4-D4	2/0/6	851.98	491633	2675
C30-V4-D4	0/4/4	208.54	98368	2797
C40-V4-D4	0/8/0	133.40	55410	2903
C50-V4-D4	0/6/2	57.65	20764	3415
C10-V8-D4	0/0/8	–	460257	–
C20-V8-D4	0/0/8	–	301734	–
C30-V8-D4	0/0/8	–	190460	–
C40-V8-D4	0/0/8	–	179803	–
C10-V16-D4	0/0/8	–	274821	–
C20-V16-D4	0/0/8	–	128255	–
C10-V4-D6	0/0/8	–	637938	–
C20-V4-D6	0/0/8	–	419442	–
C30-V4-D6	0/0/8	–	339490	–
C40-V4-D6	0/0/8	–	313328	–
C50-V4-D6	0/0/8	–	272332	–
C10-V8-D6	0/0/8	–	366596	–
C20-V8-D6	0/0/8	–	178572	–
C10-V16-D6	0/0/8	–	245182	–

Test (modKLadn)	S/U/T	Cpu	Cons	Mem
C10-V4-D4	0/8/0	2.73	2708	3007
C20-V4-D4	0/8/0	6.89	4805	3399
C30-V4-D4	0/8/0	9.08	6383	3429
C40-V4-D4	0/8/0	16.87	11012	3501
C50-V4-D4	0/8/0	20.87	14330	4027
C10-V8-D4	0/8/0	25.18	11868	6264
C20-V8-D4	0/8/0	502.22	151215	9587
C30-V8-D4	0/6/2	570.79	159546	10852
C40-V8-D4	0/3/5	655.53	164708	11309
C50-V8-D4	0/2/6	934.99	199231	12379
C10-V16-D4	0/0/8	–	497959	–
C20-V16-D4	0/0/8	–	150031	–
C10-V4-D6	0/8/0	2.18	1474	4129
C20-V4-D6	0/8/0	7.66	4534	5441
C30-V4-D6	0/8/0	14.82	7395	5628
C40-V4-D6	0/8/0	21.95	10432	6506
C50-V4-D6	0/8/0	33.71	15513	6667
C10-V8-D6	0/8/0	53.73	20948	9938
C20-V8-D6	0/8/0	172.09	38982	16164
C30-V8-D6	0/7/1	447.32	86268	20550
C40-V8-D6	0/2/6	885.67	115207	22524
C50-V8-D6	0/2/6	1113.81	208242	24986
C10-V16-D6	0/3/5	375.51	95405	24267

Fig. 4. Benchmarks using Ladner translation (`medium`)

2. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE-96*, Lecture Notes in Artificial Intelligence, New Brunswick, NJ, USA, August 1996. Springer Verlag.
3. H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *LNAI*, pages 272–275, Berlin, July13–17 1997. Springer.
4. E. Giunchiglia, F. Giunchiglia, and A. Tacchella. SAT-Based Decision Procedures for Classical Modal Logics. *Journal of Automated Reasoning*, 2000. To appear.
5. Armando Tacchella. *SAT system description. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Collected Papers from the International Description Logics Workshop (DL'99)*. CEUR, July 1999.
6. Armando Tacchella. *SAT User's Manual, 1999.
7. D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
8. Enrico Giunchiglia and Armando Tacchella. A Subset-matching Size-bounded Cache for Satisfiability in Modal Logics. In *Proc. of International Conference on Theorem Proving with Analytic Tableaux and Related Methods (Tableaux 2000)*, July 2000.