*SAT user's manual
Version 1.3
DRAFT

Armando Tacchella

Contents

1 Installing *SAT **2**
- 1.1 Requirements . . . 2
- 1.2 Preliminaries . . . 2
- 1.3 Compiling GLU . . . 3
- 1.4 Compiling *SAT . . . 4
- 1.5 Compile time options . . . 4

2 Running *SAT **7**
- 2.1 At first glance . . . 7
- 2.2 Input syntaxes . . . 9
  - 2.2.1 LISP syntax . . . 9
  - 2.2.2 C syntax . . . 10
  - 2.2.3 LWB syntax . . . 10
  - 2.2.4 TPTP syntax . . . 10
- 2.3 Command line parameters . . . 11
  - 2.3.1 Davis Putnam parameters . . . 11
  - 2.3.2 Propositional logic parameters . . . 12
  - 2.3.3 CNF conversion parameters . . . 14
  - 2.3.4 Modal logic parameters . . . 15
  - 2.3.5 Miscellanea . . . 17
- 2.4 Output data . . . 17

Chapter 1

Installing *SAT

1.1 Requirements

To complete the installation steps and run *SAT your system must be equipped with the following tools (standard in most Linux distributions):

- GNU version of the `make` utility;
- GNU version of the `m4` utility (GLU library `configure` scripts require it to work properly);
- GNU C compiler `gcc`;
- GNU automatic lexer/parser generators `flex` and `bison`;

If you do not have these tools installed on your machine, have a look at:

`http://www.gnu.org/software/software.html`

for instructions on how to download them. You might also succeed in installing and running *SAT using different tools, but this will require you to undertake the nontrivial process of modifying the installation files.

1.2 Preliminaries

In the following, assume that you have downloaded the archive `StarSAT.tar.gz` in the directory `/usr/projects`. You need to expand it first with the command:

```
> cd /usr/projects
/usr/projects> tar -xvzf StarSAT.tar.gz
```

This creates the directory `/usr/projects/StarSAT` containing the following files and directories:

`README` including contact information, a disclaimer and a short copyright notice;

`LICENSE` *SAT is distributed under the terms of the GNU General Public License

`MakeGLU` a script that takes care of building GLU;

`Manual/` the directory containing the postscript file of this manual;

`glu-1.3/` the directory containing the GLU source code;

`src-1.3/` the directory containing the *SAT source code.

1.3 Compiling GLU

Before compiling *SAT you need to build and install GLU. The script `MakeGLU` takes care of this process. All you need to do is run it:

```
/usr/projects>cd StarSAT
/usr/projects/StarSAT>MakeGLU
```

`MakeGLU` assumes that the GNU programs `make`, `m4` and `gcc` are available in the current path. If this is not the case, `MakeGLU` will not work. Whatever your local installation of the above GNU tools is (for example `/usr/gnu/bin/`) you are encouraged to add it to your `PATH` variable. To do so:

```
bash$>export PATH=$PATH:/usr/gnu/bin
csh$>setenv PATH $PATH:/ust/gnu/bin
```

At the end of the script you should find two more directories, named `include` (GLU header files) and `lib` (GLU library object code). Now you are ready to compile *SAT.

1.4 Compiling *SAT

*SAT comes with a makefile that takes care of compiling all *SAT modules and linking them into a single executable. The following commands will do:

```
/usr/projects/StarSAT>cd src
/usr/projects/StarSAT/src>make depend
/usr/projects/StarSAT/src>make
```

The `make depend` command will not generate any executable, but you are required to run it after changes to the `makefile` (see section 1.5 for more details) and for any first-time installation of *SAT. Running `make` might generate different executables according to the settings specified in the `makefile`. By default, using the `makefile` that comes along *SAT distribution, the executable `kSAT` is generated (satisfiability in multimodal logic K(m)). The default input syntax is LISP-like and closely resembles the one described in the KRSS document [Patel-Schneider and Swartout, 1993] (see section 2.2 for more details).

Whenever you need to re-build *SAT, possibly with different compile-time options, you need to run:

```
/usr/projects/StarSAT/src>make clean
```

which forces recompilation of every module with the new options. Be careful since `make clean` clears each executable so far generated as well as any `emacs` backup file (like `file.txt~`)!

1.5 Compile time options

There are some sections of the `makefile` that you might edit in order to get different compile-time options built in *SAT. Since *SAT is an on-going research project, some of its features may be not fully documented yet in this manual. They will get support as soon as they are well established.

The section in the `makefile` that you may wish to edit are:

`INPUT SYNTAX` you can choose what kind of input syntax is compiled in *SAT. There are four input syntaxes available at the moment (they are described in detail in section 2.2). For example the default KRSS-like input syntax declarations are:

```
PARSER = alc.y
LEXER = alc.lex
SYNTAX = -DNARY
```

The declarations `PARSER` and `LEXER` choose the parser and lexer modules to be loaded into the *SAT executable. The modules `alc.y`, `c.y`, `lwb.y`, `tptp.y` can be assigned to `PARSER`, and the corresponding `.lex` files can be assigned to `LEXER`. Remember that:

- the declarations `PARSER` and `LEXER` *must* coincide, modulo the extensions;
- there can be *only one* parser/lexer built into *SAT at the same time.

The declaration `SYNTAX` may be either `-DNARY` or `-DBARY`. TPTP syntax declaration bears an additional `-DPURGE_COMMENTS` that tells to *SAT to ignore comment lines in the header of the input formula. Using `-DNARY`, formulae are internally parsed as trees where `AND` and `OR` nodes have an arbitrary number of sons (`alc` and `c` modules). When `-DBARY` is used, `AND` and `OR` nodes in the parse tree do not have more than two sons (`tptp` and `lwb` modules). The `makefile` that comes along *SAT distribution has ready-to-play sets of declarations for all the four input syntaxes. You just need to comment each set of declarations, but the one that you want to use.

`DIRECTIVES` The purpose of these compile-time directives is to squeeze more performance out of the box. You can choose among different settings. The first setting, labeled `RGFK`, is best suited for random generated formulae and, specifically, for extended CNF formulae and Tableaux99/2000 benchmarks suites. The second one, labeled `HGFK`, is best suited for hand generated formulae and, specifically, for Tableaux98 benchmarks suite. Running *SAT always with the same setting will not hurt, but you might not be able to squeeze top performance out of it. There is also a "bare bones" setting that avoids all kinds of compile time add-ons in *SAT. The default setting is `RGFK`, you might choose a different one by commenting in and out the corresponding lines in `makefile`.

`TARGET EXECUTABLE` four different executables might be generated during the build process:

- `kSAT` Satisfiability in logic K(m)
- `kPROVE` Provability in logic K(m) (the input formula is negated)
- `eSAT` Satisfiability in logic E(m)
- `ta4eSAT` Translational approach for satisfiability in logic E (the input formula is translated to an equivalent satisfiability test in K(2))

The `makefile` that comes along *SAT distribution has ready-to-play sets of declarations for all the four target executables. You just need to comment each set of declarations, but the one that you want to use.

Chapter 2

Running *SAT

2.1 At first glance

If you run any of the executables generated by the `makefile` (see section 1.5) without specifying any command line parameters you get a help banner describing *SAT usage. The first few lines look as follows:

```
> kSAT

Welcome to *SAT, a platform for experimenting with SAT-based procedures!
Usage: kSAT [<Options>] <Input_file>
...
```

More lines of explanation follow and we will go thru them in the section 2.3. In the remaining of this section we are going to outline a simple example using the default syntax and options.

We start with writing the file `wff.alc`[1] that contains just the following expression:

```
(NOT (IMP (AND (ALL R0 (IMP C0 C1)) (ALL R0 C0)) (ALL R0 C1)))
```

corresponding to the formula $\varphi = \neg((\Box_0(c_0 \supset c_1) \wedge \Box_0 c_0) \supset \Box_0 c_1)$. Clearly, φ is not satisfiable in logic K. In order to have `kSAT` prove this fact, all we need to do is to type:

[1] This file can also be found in `StarSAT/src-1.3`

```
>kSAT wff.alc
```

and we get an answer that looks like:

```
Sat, Btime, Dtime, ConsChecks, FConsChecks, EPcalls, EPbackts, Unit,
Pure, MSplit, DSplit, TSplit, Assign, Backts, AddVar, Macc, Msucc,
MaTime, Mstore, Mtrash, MsTime, Dacc, Dsucc, DaTime, Dstore, Dtrash,
DsTime, MaxMem, CacheMem
0, 0.01 sec, 0.00 sec, 1, 1, 0, 0, 7, 0, 0, 0, 0, 1, 0, 2, 0, 0, 0, 0,
0.00, 0, 0, 0, 0 ,1, 0.00, 0, 135, 3
```

meaning that the formula is unsatisfiable (`0` in the field `Sat`). The remaining data regards *SAT run time, number of consistency checks, etc. The meaning of these output values will be explained later in section 2.4. *SAT prints a compact description of the output data by default. This is adequate when collecting data from large scale experimentation but a more "human-oriented" output can be obtained with the `-r2` option ("enRich" output). The command:

```
>kSAT -r2 wff.alc
```

yields the following output:

```
The formula was unsatisfiable
--------------------TIMINGS
Parse time        0.00 sec
Preprocess time   0.00 sec
Translate time    0.00 sec
Build time        0.00 sec
Decision time     0.00 sec
--------------------GENERAL
Consistency checks         1
Failed consistency checks  1
Early Pruning (calls)      0
Early Pruning (backtracks) 0
--------------------DAVIS PUTNAM
Unit propagations  7
Pure literals      0
Splits             0
  on modal atoms   0
  on added var.    0
```

```
Assignments found  1
Bactracks          0
--------------------CNF CONVERSION
Added variables   2
--------------------CACHING
Cache accesses      0
Cache successes     0
Total search time   0 (msec)
Items stored        0 (trashing ratio 0.00)
Totale store time   0 (msec)
Dependencies tried  0
Dependencies found  0
Total search time   0 (msec)
Dependecies stored  1 (trashing ratio 0.00)
Total store time    0 (msec)
--------------------MEMORY
Maximum memory allocated by SATO             135 Kb
Total memory allocated for caching           3 Kb
```

2.2 Input syntaxes

We start our journey into *SAT world talking about the input syntaxes that might be used to present problems to the system.

2.2.1 LISP syntax

This syntax was used in the very first implementation of KsatLisp ([Giunchiglia and Sebastiani, 1996]) the LISP ancestor of *SAT. It is derived from the basic KRSS syntax ([Patel-Schneider and Swartout, 1993]) for knowledge representation. The grammar outline follows:

```
formula      ::= boolean_expr |
                 rule_expr    |
                 atomic_expr;
boolean_expr ::= (unary_op formula) |
                 (binary_op formula formula) |
                 (nary_op formula formula_list);
formula_list ::= formula form_list | ϵ;
rule_expr    ::= (rule_op rule formula);
atomic_expr  ::= atom | TOP | BOTTOM;
```

```
unary_op     ::= NOT;
binary_op    ::= IFF | IMP;
nary_op      ::= AND | OR;
rule_op      ::= ALL | SOME;
atom         ::= C0 | C[1-9][0-9]*
rule         ::= R0 | R[1-9][0-9]*
```

2.2.2 C syntax

This syntax was used by the first C implementation of Ksat, KsatC. *SAT retains this syntax for backward compatibility. This is also the default syntax used by the formula generator that you can download from *SAT page. Files are generated according to the syntax:

```
formula      ::=   boolean_expr |
                   rule_expr    |
                   atomic_expr;
boolean_expr ::=   boolop ( formula other ) |
                       - boolop ( formula other );
other        ::= formula other | ϵ;
rule_expr    ::=  boxop agent ( formula ) |
   - boxop agent ( formula );
atomic_expr  ::= [1-9][0-9]* | -[1-9][0-9] | T | F;
boxop        ::= #;
boolop       ::= v | ^;
agent        ::= (0|[1-9][0-9]*)
```

2.2.3 LWB syntax

*SAT reads LWB syntax for modal formulae. This syntax was introduced in the LWB system ([Heuerding *et al.*, 1996]) and used in Tableaux98 competition benchmarks ([de Swart, 1998]). The rules used by *SAT are more strict than what is allowed by LWB, since they require *all* formulas to be parenthesized, even those beginning with the negation symbol.

2.2.4 TPTP syntax

This syntax has been first introduced in Tableaux99 ([Massacci, 1999]) competition. *SAT understands this syntax to its full extent, except the “axiom” declarations since, at the moment, *SAT is not able to deal with axioms.

2.3 Command line parameters

*SAT behavior can be influenced by several command line parameters. A short explanation about such parameters is given by the *SAT help banner. At the end of the banner, the default value for each parameter described in this section is displayed. In this section we elaborate a little on the short descriptions given in the banner.

2.3.1 Davis Putnam parameters

In this subsection we discuss about the parameters that influence directly the behavior of the embedded Davis-Putnam algorithm and that can be used in the context of modal logics. There are more parameters that affect the embedded propositional solver, but not all of them might be used in *SAT when reasoning in modal logics (they are discussed next, in subsection 2.3.2). In the current version of *SAT, the embedded Davis-Putnam solver is SATO 3.2 ([Zhang, 1997]).

We are now concerned with the parameters described in the following portion of the help banner:

```
********** SPLITTING **********
   -s<n>    : choose a different splitting heuristic (n = 1..6):
               n = 1 MOM strategy to all active variables
               n = 2 MOM strategy to all active variables in the
                     shortest clauses
               n = 3 choose the min variable in the first shortest
                      clause (sign = CHOICE1)
               n = 4 choose a variable in a shortest clause using
                     occurrence as the second criterion (sign = CHOICE1)
               n = 5 choose a variable with the highest occurrence
                     (sign = CHOICE1)
               n = 6 choose a variable with the highest occurrence
                     (sign = ?)
   -l<n>    : choose a different MOM strategy (n = 1..3 - 11..13 - 21):
               n < 11       original SATO weighting:
                   n=1 sign =(neg_count>pos_count)? CHOICE1 : CHOICE2;
                   n=2 sign = FF
                   n=3 sign = TT
               11 <= n < 21 clauses length does not matter (sign as n - 10)
               n = 21       Boehm's weight and sign
   -v       : splitting bias becomes CHOICE1 = TT, CHOICE2 = FF
```

Parameters `-s<n>` and `-v` are inherited from the original SATO solver, while `-l` has been added in the *SAT embedded implementation of SATO.

The parameter `-s<n>` allows you to choose among several splitting heuristics, i.e., the way SATO chooses the atom on which disjunctive branching occurs. The default value for this parameter is `-s2`, that is the well-known MOMS heuristic: the literal with maximum occurrences in clauses of minimum size is chosen. Although MOMS is by far the most effective heuristic among those available for propositional logics, the same might not be true for modal reasoning.

In the context of MOMS strategies (`-s2` choice), it is possible to rank variables in several ways and, also, the choice of the splitting sign might be different. Parameter `-l<n>` serves this purpose, in that it alters the way SATO weights the variables and chooses the splitting sign. The default is `-l1`, i.e., atoms appearing often in binary clauses are preferred. If the most occurring sign is positive, then splitting happens with negative sign and vice versa. This has the clear purpose of maximizing unit propagations and thus cutting through the search tree quickly. Again, this is one of the most efficient heuristics in propositional logic, but this might not be true in the context of modal reasoning. This is why, besides the original one, we provide different ways to choose weight and sign. For example, with `-l<n>` and `n` in between 11 and 13, if the pool of selected clauses does not contain binary clauses, atoms are weighted according to their occurrence rate (not flatly, as it might happen in SATO!). Further improving on this line, `-l21` emulates Boehm splitting heuristics, which "prefers" atoms occurring in binary clauses, but also considers occurrences in clauses of bigger size to break the ties. The sign is chosen accordingly to the highest number of occurrences, i.e., the sign is going to be positive if occurrences of the chosen variable bear the positive sign more often than the negative sign, and vice versa.

Finally, `-v` parameters allows you to "swap" the splitting sign selected. The meaning of this option depends largely on the value of `-s<n>` and `-l<n>`. By default, the first choice is "false", the second is "true".

2.3.2 Propositional logic parameters

The parameters described in this subsection affect the behavior of *SAT in a way that is not always harmless for reasoning in modal logics. They enforce optimizations that might lead SATO to generate incomplete sets of assignments. Since our algorithms rest on the assumption that all the propositional assignments can be possibly enumerated, enabling some of these options in the context of modal logics *yields unsound decision procedures*. Why we decided to leave them in spite of this danger? Since *SAT features an internal space-efficient CNF conversion (see next section 2.3.3), it might be used as a non-CNF propositional solver. In this case all the optimizations described in this subsection may come into play, effectively boosting up performance. The parameters we are now interested in are displayed in the *SAT help banner as follows:

```
********** OPTIMIZATIONS (PROPOSITIONAL LOGIC ONLY!!)  **********
   -p<n>    : enable pure literal detection
              n=1 on added variables only
              n=2 on any variable
   -g<n>    : enable propositional backjumping
   -h       : enable relaxation to horn clauses
```

The `-p<n>` switch enables pure literal detection and simplification. The default value for this option is `1`, i.e., pure literal detection is performed on variables added by the optimized CNF conversion (which is also a default). Pure literal rule would be dangerous if applied to modal atoms, i.e., atoms masquerading a modal formula. For the sake of efficiency, it can be applied to propositional atoms. In particular, we chose to make it available on added variables, i.e., on variables added during a CNF conversion (see subsection 2.3.3 for more details). Anyway you do not need to alter `-p<n>` value, because the optimized CNF conversion automatically changes the value of `-p<n>` to `1`.

The second (`-g<n>`) and third (`-h`) parameters *must not* be enabled, except when using *SAT as a non-CNF propositional decision procedure. They are both disabled by default. `-g<n>` enforces backjumping, with stored reason of length `n` and this might disrupt *SAT behavior when backtracking from *modal* inconsistencies. `-h` enforces relaxation to horn clauses, i.e., literals occurring in horn clauses only are never picked up, but horn clauses are subject to evaluation and unit propagation. When the original formula is reduced to a set of horn clauses without finding contradictions, SATO halts with a satisfiability result. This behavior actually prevents SATO from generating the complete sets of assignments.

2.3.3 CNF conversion parameters

*SAT features an internal CNF conversion that allows it to deal with modal formulae even if they are not in extended CNF. Internal conversion is necessary since SATO deals with sets of clauses only. The help banner output is:

```
********** INTERNAL CNF CONVERSION **********
   -k       : choose the conversion type
              n=0 standard CNF conversion
              n=1 optimized CNF conversion (also triggers -n, -p1)
   -n       : pushing down negations
   -d       : don't prefer splitting on independent variables
```

The first parameter, `-k<n>` allows you to choose among two different CNF conversions, a basic one (`-k0`) and an optimized one (`-k1`, the default). The optimized CNF conversion is essentially an implementation of what described in [Plaisted and Greenbaum, 1986]. Both conversions try to keep the number of added variables small, the second one also keeps the clause count small. In order for it to be even more effective, `-k1` triggers also `-n` parameter (described next) and `-p1`, with the result that the propositional search is less burdened by added variables.

Parameter `-n` forces *SAT to preprocess formulae by pushing down negations. The following rewrite rules are recursively applied:

$$\begin{array}{rcl} \neg(\varphi \wedge \psi) & \Rightarrow & \neg\varphi \vee \neg\psi \\ \neg(\varphi \vee \psi) & \Rightarrow & \neg\varphi \wedge \neg\psi \\ \neg(\varphi \supset \psi) & \Rightarrow & \varphi \wedge \neg\psi \\ \neg(\varphi \equiv \psi) & \Rightarrow & \varphi \equiv \neg\psi \\ \neg\Diamond\varphi & \Rightarrow & \Box\neg\varphi \end{array}$$

By default, these rules are enforced because of the default value of the `-k` switch. *Independently* of the `-n` switch, the following rules along with trivial propositional simplifications are recursively applied:

$$\begin{array}{rcl} \Diamond\varphi & \Rightarrow & \neg\Box\neg\varphi \\ \varphi \supset \psi & \Rightarrow & \neg\varphi \vee \psi \end{array}$$

Both CNF conversions may deal with formulae that are not preprocessed with `-n`, but their results are somewhat less compact and they may end up confusing SATO a lot more than what is necessary.

2.3.4 Modal logic parameters

*SAT features some optimizations that speed up modal reasoning. The help banner describes them as follows:

```
-e<n>    : modal early pruning: 0 disable, 1 enable
-b<n>    : enable modal backjumping: 0 disable, 1 enable
-m<n>    : with n<=3 enable caching with a bit matrix:
            n=1 enable caching of satisfiable subtests
            n=2 enable caching of unsatisfiable subtests
            n=3 enable both kinds of caching (2+1)
           with n>3 enable caching with a hash table:
            n=4 enable caching of satisfiable subtests
            n=5 enable caching of unsatisfiable subtests
            n=6 enable both kinds of caching
-w<n>    : cache memory size for a bit matrix (a multiple of 32 is good!)
```

Both `-e` (enabled by default) and `-b` (disabled by default) try to reduce the search space spanned by SATO using information that comes from modal successors. Both heuristics might be used to limit *trashing* of propositional assignments. Trashing happens when the propositional solver repeatedly discovers the same modal inconsistency, i.e., when the propositional backtracking does not eliminate modally inconsistent kernel(s) from the assignments spanned. Although modal early pruning and modal backjumping are not part of the core algorithm, using *SAT with none of these options yields an enormous increase in the CPU time needed to solve formula instances.

Modal early pruning (`-e<n>`) is a lookahead technique that tries to eliminate modal inconsistencies as soon as they arise in the assignment built by the propositional solver. Before every disjunctive branch, the assignment so far generated is checked for modal inconsistencies. If the incomplete assignment is already inconsistent then backtracking occurs. Modal backjumping techniques (`-b<n>`) come into play only after a modal inconsistency is discovered in a complete assignment. The propositional solver is then forced to neglect trivially irrelevant nodes, i.e., nodes that were not responsible for the failure of the last consistency check. Such technique is simple and fast since, as opposed to `-e`, it does not require any additional consistency check. Although backjumping and early pruning could be both enabled in principle, it is not safe to do so in the current implementation of *SAT.

Caching is a key feature when dealing with big formula instances, where the same subproblem might occur more than once. Currently *SAT features two forms of caching:

- a *subset-matching* cache, i.e., *SAT can query for subsets (supersets) of the stored sets of formulas. In the current implementation, this cache (a bit matrix) is also bounded in size: only some results are stored and when the cache is full, former results are discharged using a FIFO politic.

- an *exact-matching* cache, i.e. *SAT is ensured not to repeat the very same test over and over again. In the current implementation, this cache (a hash table) is unbounded in size: all the results are stored and the space allotted is not guaranteed to stay compact.

The default choice is -m3, i.e., bounded subset matching cache storing both satisfiable and unsatisfiable subtests. The choice of a size bounded cache as a default rests on the fact that caching partial results in many interesting modal logics requires exponential space in the worst case. *SAT bounded caching mechanism guarantees that space requirements grow polynomially with the input size. The assumption is that partial results are reused only "locally", i.e., the more the assignments generated by the propositional solver differ, the less the probability of previously cached results being reused. In all the cases in which this is true, a cache that forgets results with a FIFO (First In, First Out) mechanism can effectively trim down the search time. By default, *SAT forgets the oldest result as soon as the space alloted is exhausted, and it keeps forgetting previously cached results in the same order in which they were stored. The actual size of the cache depends on the command line parameter -w<n>. Tuning the value of the cache size is a matter of finding the right balance between access time and cache persistence. A small cache is faster but less persistent, since it quickly overwrites former results as new ones are pumped in. A big cache is slower but more persistent, since old results have more chance to survive. The parameters -m<n> and -w<n> make sense with the executables kSAT, kPROVE, ta4eSAT since they work for modal logic K. The executable eSAT features its own caching algorithm (appropriate for classical logics). In eSAT caching is not an option since it is part of the algorithm. As a consequence, -m<n> will not have any impact on the behavior of eSAT.

2.3.5 Miscellanea

This section describes the parameters controlling miscellaneous features of *SAT. The help banner describes them as:

```
********** MISCELLANEA **********
   -r<n>     : determines output level:
                n=0 single line data without comments
                n=1    "     "  "   with comments
                n=2 verbose output
   -o<sec>   : enable a timeout (disable any timeout if 0)
```

The option `-r` influences the form in which *SAT outputs its result. As anticipated in section 2.1 *SAT can provide the user with several kinds of output. A single line output (`-r0`) that lists the satisfiability result (`0` or `1`) followed by some of the data recorded during the execution of *SAT (e.g., decision time, recursive calls, allocated memory). With `-r1` (the default) a line of comments appears before the data is displayed. With `-r2` switch it is possible to obtain longer and more informative reports of *SAT runs. Besides comments accompanying every data, the "rich" report includes time for parsing, preprocessing and building the internal data structure, which are not present in the "short" description. If *SAT reaches a time out, the report will be printed with the data recorded before the timeout was reached.

The option `-o<n>` sets a timeout in seconds of CPU time. The default value is `1000`, i.e., about twenty minutes of CPU time. The timeout limits the internal build time (and thus CNF conversion) and the core decision time. Parsing, preprocessing and normalization times are not taken in account.

2.4 Output data

When *SAT terminates the decision process, it prompts the user with a report, possibly coming in different forms (see section 2.3.5) containing data about the resources allocated (time, space) and data about the algorithm (e.g. number of consistency checks, disjunctive branches, unit resolutions). In the following, the output data is described in the same order in which it is displayed. Notice that display order in the short (single and double line) output corresponds to the rich output. For the sake of clarity, output information is grouped as follows:

Result: in the short output either `0` (unsatisfiable) or `1` (satisfiable). In the rich output report the result is described lengthy (e.g. `The formula was satisfiable`).

TIMINGS: `Parse time` is the time taken to read and parse the input formula. `Preprocessing time` includes time needed for input normalization, i.e. (possible) conversion of binary formula trees into n-ary ones, pushing down of negations, recursive application of the rule $\Diamond\varphi \Rightarrow \neg\Box\neg\varphi$ (see 2.3.3). If the executable is `ta4eSAT` then `Translate time` accounts for the time taken in translating a problem from logic E to $K(2)$. `Build time` includes the time needed to build the internal data structure, to replace modal formulae with atoms, to build the cross-reference table and to initialize the SAT algorithm (CNF conversion is part of this process). Finally, `Decision time` is the time taken to solve the formula. In the single-line output description, this time and the build time are the only values displayed.

GENERAL: `Consistency checks` is the number of modal consistency checks performed by *SAT, `Failed consistency checks` is the fraction of such checks that failed, i.e., forced the propositional solver to look back for another assignment. If early pruning is enabled, `Early pruning (calls)` accounts for the number of early consistency checks, `Early pruning (backtracks)` is the fraction of such checks that was inconsistent, i.e. the number of times that early pruning prevented the propositional solver from exploring irrelevant branches.

DAVIS PUTNAM: `Unit propagations` accounts for the total number of unit resolutions (unit clauses detected and resolved). `Pure literals` accounts for the total number of applications of the pure literal rule. In modal logics, this must be always `0` unless optimized CNF conversion (option `-k1`) is selected. `Splits` is the total number of disjunctive branches performed by the propositional solver, `on modal atoms` is the fraction of such branches that happened on modal formulae, `on added var.` is the fraction of such branches that happened on variables added by the CNF conversion (this might be non-zero even if `-d` parameter is not set). `Assignments found` is the total number of assignments enumerated by the propositional solver. `Backtracks` accounts for the total number of *propositional* backtracks, i.e., backtracks caused by some purely propositional inconsistency.

CNF CONVERSION: `Added variables` is the number of variables added by the CNF conversion.

CACHING: caching data is displayed only when some kind of caching (option `-m<n>`) is enabled. Caching may concern satisfiability results or unsatisfiability results (dependencies). In display order, `Cache accesses` is the number of times that the cache for satisfiability results is accessed, `Cache successes` is the fraction of such accesses that was successful. `Items stored` is the *total* number of results stored. Since the cache is bounded some of these results were possibly trashed (i.e., overwritten). `trashing ratio` gives the ratio between trashed assignments and cache size. The higher the trashing ratio, the smaller the cache persistence and vice versa. `Total search time` and `Total store time` give the total time spent for searching and storing satisfiable subtests, respectively. `Dependencies tried` is the number of times that the cache for unsatisfiability results is accessed, `Dependencies found` is the fraction of such accesses that was successful. As in the case of satisfiability caching, `Dependencies stored` and `trashing ratio` are, respectively, the number of unsatisfiability results stored and the measure of dependency cache persistence. `Total search time` and `Total store time` give the total time spent for searching and storing dependencies, respectively.

MEMORY: *SAT decides satisfiability of a formula by implicitly spanning a modal structure in a depth-first fashion. The relevant parameter is thus the peak allocation of memory, i.e. the maximum memory allocated for SATO data structures on any search path. In display order, this is the first data of this section. If caching is enabled then also the space alloted for caching partial results is displayed.

Bibliography

[de Swart, 1998] H. de Swart, editor. *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, number 1397 in Lecture Notes in Artificial Intelligence. Springer-Verlag, May 1998.

[Giunchiglia and Sebastiani, 1996] F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In *Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning - KR'96*, Cambridge, MA, USA, November 1996. Also DIST-Technical Report 9607-08 and IRST-Technical Report 9601-02.

[Heuerding *et al.*, 1996] A. Heuerding, G. Jager, S. Schwendimann, and M. Seyfried. The Logics Workbench LWB: A Snapshot. *Euromath Bulletin*, 2(1):177–186, 1996.

[Massacci, 1999] Fabio Massacci. Design and results of the Tableaux-99 Non-classical (Modal) Systems comparison. In Neil V. Murray, editor, *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-99)*, volume 1617 of *LNAI*, pages 14–18, Berlin, June 07–11 1999. Springer.

[Patel-Schneider and Swartout, 1993] Peter Patel-Schneider and Bill Swartout. Description-logic knowledge representation system specification, 1993.

[Plaisted and Greenbaum, 1986] D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[Zhang, 1997] 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.