SATbed User Documentation*

(Version 0.70, to be released 31 July 2003)

Matthias F. Stallmann, Franc Brglez, and Xiao Yu Li

{mfms,brglez,xyli}@unity.ncsu.edu

Summary. This document gives an overview of how to use the SATbed software for managing various aspects of experiments with satisfiability solvers. The encapsulated solvers whose output is post-processed by SATbed include chaff [MMZ+01], OpenSat [LeB03], QingTing (two versions) [LSB03], sato [Zha97], UnitWalk (versions 0.944 and 0.98) [HK01, HK03], and WalkSat [SK02].

Background Materials. (included with the SATbed release)

- A SATbed tutorial in PowerPoint (file `2003-SATbed-Brglez`).
- The revised conference paper [BSL03] in PDF (file `2003-SAT-Brglez.pdf`).

Contents

1 Installation 2

2 The Configuration File 3

3 Details of Each Stage 6

4 Utility Programs 10

5 Configuration Files for Demonstration 10

*last revision July 29, 2008

1 Installation

SATbed comes in the form of an archive (.zip or .tar.gz). To install it, unpack the archive in a directory of your choice, go into the SATbed_v0.*xx* directory and run the INSTALL script. This will compile a variety of programs and put them into the bin subdirectory. You will be asked whether or not you want to install each of the available solvers (details on these are provided later). Typical output for the INSTALL script is shown in Figure 1.

Immediately after installation, it is recommended that you execute the script demo-7.cfg in the demos subdirectory. You can do this by typing

```
bin/@SATbed demos/demo-7.cfg
```

Your results, stored in the directory demo_outputs, should match those in demo-7_ARCHIVE, except for runtime details. The actual terminal output (which you can capture using >& *fileName*) should match what is in file demo-7.log in directory demo-7_ARCHIVE.

```
Installing SATbed binaries, version 0.60.
gcc -c -Wall -O3 cnf.c
gcc -c -Wall -O3 randomNumbers.c
gcc -c -Wall -O3 rotate_formula.c
gcc cnf.o randomNumbers.o rotate_formula.o -o rotate_formula
gcc -c -Wall -O3 verify.c
gcc cnf.o verify.o randomNumbers.o -o verify
g++ -c -Wall -O3 cnfTrim.cpp
g++ cnfTrim.o -o cnfTrim
g++ -c -Wall -O3 cnf3sat.cpp
g++ cnf3sat.o -o cnf3sat
cp rotate_formula verify cnfTrim cnf3sat ../bin\
; rm -rf *.o *~ rotate_formula verify cnfTrim cnf3sat
Install QingTing1 (y/n)? n
Install QingTing2 (y/n)? n
Install QT_methodB (y/n)? n
Install sato_modified (y/n)? n
Install UnitWalk1 (y/n)? n
Install UnitWalk2 (y/n)? n
Install walksat (y/n)? n
Install zchaff (y/n)? n
```

Figure 1: Typical output from running the INSTALL script.

2 The Configuration File

Syntax. To execute a SATbed configuration file, run the @SATbed script with the name of the configuration file as its only command-line argument.

Comments in the configuration file begin with # and continue to the end of the line. Blank lines are ignored. All other lines are *statements* of the form *identifier* = *value* (the = must be preceded and followed by white space to distinguish it from other = signs that may occur in program arguments). The *identifier* is any string with no whitespace. The *value* is any sequence of whitespace-delimited strings called *arguments*. Some arguments may take the form of comma-delimited lists.

The statements are interpreted according to context. They are used to define the locations of data and programs, to determine what scripts to run, and to give additional information important to various stages of an experiment. See Section 5 for an example.

Directory Structure.

One set of statements is used to declare the data directories for the experiments. These are optional. By default, a SATbed installation starts with a *root directory* (usually called SATbed). Below the root are subdirectories bin (programs and scripts), doc (documentation), and src (source code for programs). If data directories are not declared, SATbed will automatically create subdirectories benchmarks and results to store problem instances and solver outputs, respectively.

Data directories are declared by specifying BENCHMARK_DIR and RESULTS_DIR, respectively. For example:

```
BENCHMARK_DIR = /afs/eos/project/cbl-exp/benchm_SATcnf
RESULTS_DIR = SatExperimentResults
```

The paths may be absolute, as with the first example, or relative to the root directory, as with the second.

Each stage includes specifications of the form *programID* = *program*, as is described later. The value *program* may have any number of arguments, but the first of these always specifies a path to an executable file. There are three possible interpretations for this path:

- A path containing only the name of a file (no directory) is interpreted as the name of an executable in the bin subdirectory (additional programs, such as solvers, can be installed there).
- A relative path (one that does not begin with /) is interpreted as a path relative to the root directory. For example, there may be executable scripts used only for demonstration purposes in a subdirectory called demos.

These can be referenced as `demos/`*prog*, where *prog* is the name of the program.

- An absolute path (begins with `/`) is interpreted as the full path to the program.

Control Flow. There are six stages of execution in an experiment.

1. `REFGEN` is when reference instances, single benchmark instances, are generated (if necessary).
2. `CLASSGEN` creates classes of "equivalent" (with respect to structure and/or hardness) instances from each reference instance.
3. `SOLVER` runs one or more solvers on a specified collection of references and classes and stores output files in subdirectories below `RESULTS_DIR` (or `ROOT_DIR/results` if not specified).
4. `POST_PROCESS` interprets each raw output file created by a solver and produces two types of secondary output files (plus coresponding HTML and/or LaTeX tables): (a) *tabulated output* (default extension is `.tab`) in which the output of each instance in a class is recorded as a row in a standardized table, and (b) *statistical output* (default extension is `.stat`) in which a statistical summary gives mean, median, standard deviation, etc., of the runtime or some other measure. Other files useful for assessing the sampling error or plotting the solvability function are also produced in this stage.

   Satisfying solutions reported by solvers are verified during this stage and the outcome of the verification is recorded in the tabulated output. The SATbed user, when installing a new solver, is only responsible for supplying a program or script that creates the initial tabulated output (as discussed in more detail later).
5. The `STATISTICS` stage creates custom statistical output designed by the SATbed user. Each output file created in this stage is derived from a standardized table (`.tab` file) produced during the previous stage but need not conform to any standards in turn (unless the user wants to use such files as input during the next stage).
6. The `VIEWER` stage includes any program that collects data about more than one solver and/or more than one class and reorganizes, formats, or displays this data in a user-friendly form (e.g. LaTeX tables, tables for creating various kinds of charts with a spreadsheet, etc.).

A given run typically does not include execution of all six stages. At some point new classes might no longer be created while experiments with new solvers continue, for example. Or the progression needs to be interrupted to perform

some other operation, such as moving data to a different machine in order to run a particular solver.

The identifiers `START` and `STOP` allow the user to specify the stage at which processing begins and ends, respectively. For example, the directives

```
START = SOLVER
STOP = VIEWER
```

cause only the last four stages to be executed.

The name of a stage is used as an identifier whose value is a comma-separated list of *programID*'s of programs to be executed in that stage. For example,

```
SOLVER = chaff,sato,unitwalk
```

causes programs associated with ID's `chaff`, `sato`, and `unitwalk` to be executed. The actual paths of and some of the arguments to these are specified using the *programID* as an identifier, as in `chaff = /usr/local/bin/zchaff`.

As will become clear in the next section, the *programID*'s play important roles other than being short names for the programs they refer to. They may serve as (parts of) names of files or directories or as labels in tables of experimental results.

Families, Instances, and Classes. To simplify discussion of what happens in detail during each stage, imagine that a collection of related reference instances are organized into a *family*. One such family in our experimental work is that of medium-size satisfiable scheduling instances, called `sched_medium_s`. This includes five instances ranging from `sched03s_v00095` (with 95 variables) to `sched07s_v01386`. Each instance may have had one or more classes formed from it. For example, the classes `sched03s_v00095_P` and `sched03s_v00095_PC` each have 32 instances in addition to the reference `sched03s_v00095` from which they were created.

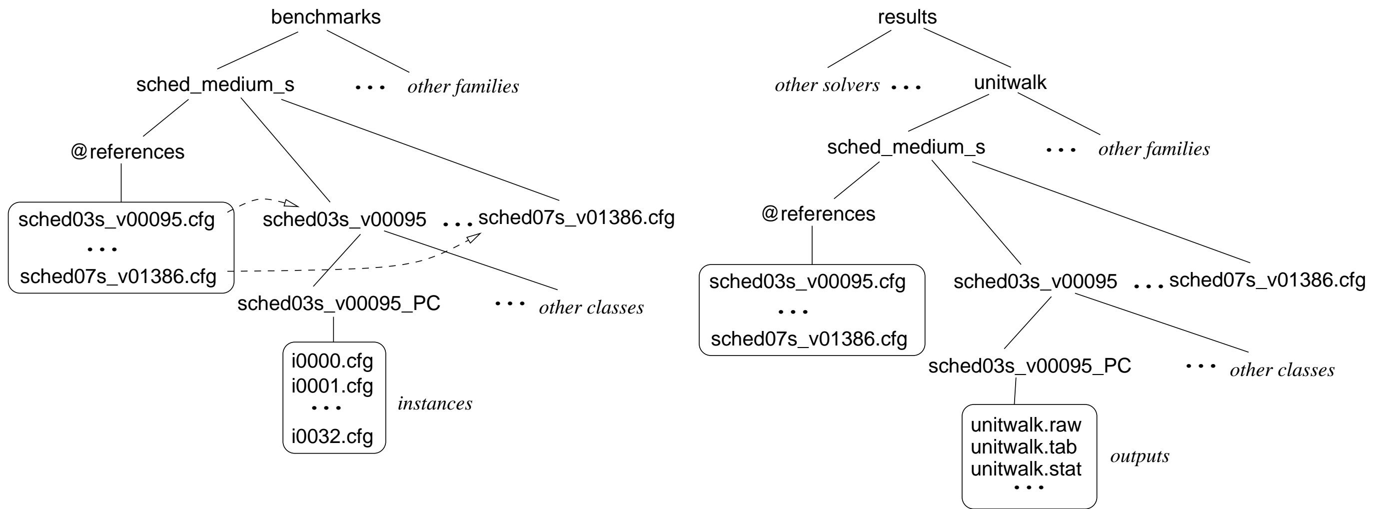
Figure 2 shows the default structure of directories used by SATbed.

When the value of an identifier specifies a list of classes (as is the case with `SOLVE_CLASSES` and `VIEW_CLASSES`, described later) each item in the list falls into one of three cases:

1. a *family* name (such as `sched_medium_s`) — refers to all classes generated from all instances in the family; when appropriate, the collection of reference instances is also treated as a class

2. an *instance* name (such as `sched07s_v01386`) — refers to all classes generated from that instance

3. a *class* name (such as `sched03s_v00095_PC`) — refers only to that single class



The benchmark directory has a subdirectory for every instance family. These, in turn, each have a `@references` subdirectory containing the benchmark instances that comprise the family and a separate subdirectory for each benchmark. The directory for a benchmark contains subdirectories for the classes derived from that benchmark. Finally, each class contains randomly sampled instances labelled i*xxxx*, where *xxxx* is a four-digit zero-padded version of the instance number. The results directory contains a subdirectory for each solver. Each solver directory, in turn, is the root of a tree that replicates the benchmark directory, or those parts of it containing instances that have already been solved by the solver.

Figure 2: The directory structure for SATbed benchmarks and results.

3 Details of Each Stage

A program for any given stage must abide by certain conventions described here. These may differ depending on the particular stage. Also, there may be additional values that need to be specified for programs in a particular stage. What follows is a summary of the important considerations for integrating programs with each stage.

Generally, the programs for a stage require specific command-line arguments, which must appear last on the command line. If additional arguments are desired there are two ways to incorporate them. One is to write the program in such a way that the additional arguments can be specified before the required ones and specify them as part of the value of the *programID*. For example,

```
sched_medium_s = sched_classic.tcl 3 7 yes
```

specifies that, to generate the family `sched_medium_s`, the instance-generating script `sched_classic.tcl` is run with three arguments, the first two specifying lower and upper bounds on instance size, and the third specifying satisfiable

Table 1: Command-line arguments and extra information needed for each stage.

stage	program arguments	other info
REFGEN	*refDirPath*	none
CLASSGEN	*refPath seed*(3 ints) *instPath*	SEED [CLASS_SIZE]
SOLVER	*instance timeLimit output*	SOLVE_CLASSES
POST_PROCESS	*output timeLimit table*	none
STATISTICS	*table*	none
VIEWER	*fileWithListOfPaths*	VIEW_SOLVERS,VIEW_CLASSES

instances. A fourth argument, giving the path to a directory containing the instances, is added by the `@SATbed` interpreter.

The second way to specify additional arguments is to supply a "wrapper script" for a program so that the script only reads the required arguments from the command line and supplies the others internally. This approach is also useful when the arguments required by a stage are not in the same form as those required by the program (which may be a solver down-loaded from elsewhere). For example, the `unitwalk` solver is run from the command line as follows:

`unitwalk -f` *cnfInput* `-t` *timeLimit* `>>` *resultsFile*

but SATbed requires three command-line arguments listed with no intervening directives. A script that does the required conversion is easily written.

Table 1 summarizes the program arguments and additional information needed in each stage. More details are given below.

REFGEN. A program in this stage takes one command-line argument, the directory where the references are to be generated. This directory has already been created when the program is run and its path, relative to the benchmark directory, is *programID*`/@references`. The *programID* also serves as the name of the family formed by these references.

CLASSGEN. A program in this stage has three arguments. The first is a path to an input (reference) instance. The second is a comma-delimited list of three short integers giving a random seed for an IEEE standard 48-bit random number generator (available with most C compilers). The third argument is an output file name — the instance name with a `.cnf` extension (the program will be executed in the directory containing class instances). The program is responsible for generating exactly one random instance of the desired class, which will be stored in the output file.

Two other important conventions need to be observed in order to ensure repeatability of experiments:

1. A starting seed must always be specified in the configuration file, as in

`SEED = 1989,1863,1776` (numbers are separated by commas).

2. Every output file created by a `CLASSGEN` program must begin with two lines of comments, the first starting with the initial seed, the second with the final seed, both in the same comma-delimited format as required for the second argument of the program. This allows for the recreation of any part of the random sequence of instances generated.

The default size for a class is 32 instances (not including the reference instance, which is always installed as instance 0). The class size can be changed by giving a value to the `CLASS_SIZE` identifier.

SOLVER. A program in this stage has three arguments. The first is the path to an input instance, the second is a time limit (in number of seconds) after which the solver "times out", and the third is the path to an output file (called a *raw* output) to which the solver *appends* information about its execution on the current instance.

The `SOLVE_CLASSES` identifier specifies the set of classes from which the instances are drawn. Its value is a list of names, each of which specifies either a whole family, all classes derived from a single reference (if the name of the reference is given), or just a single class.

POST_PROCESS. Several programs are invoked for each new raw output file during this stage. The only one that a SATbed user needs to be concerned about is a specific post-processor for each solver. The *programID* for a post-processor is always the *programID* of the solver with `_pp` appended. This ensures that the two are matched properly during SATbed execution.

A post-processor has three arguments: raw output file, time-out limit, and tabular output file. Our installation provides post-processors (in the form of Tcl scripts) for several leading solvers, with names *solverID*`_postProcess.tcl`. Someone familiar with the output format of a solver needs to write the part of the post-processor that parses the raw output.

The tabular output file, called *solverID*`.tab`, may contain arbitrary amounts of numerical data for each instance, but is required to observe the following conventions (so that the remaining programs in this and subsequent stages will work properly). Each row of the table, except for the first, header, row, corresponds to a single instance within the class. Columns are delimited by tabs and have the following special meanings.

- Column 1 contains the name of the instance, e.g. `i0003.cnf`.
- Column 2 specifies whether the instance was satisfiable, unsatisfiable, or unsolved (e.g. timed out before finding a solution, abnormal termination), using the specific identifiers `--sat--`, `--unsat--`, and `--unsolved--`.

- `Column 3` needs to have the word `verSkip` in it (verification skipped). Later during post processing a verifier program will check the solutions reported for satisfiable instances and replace the `verSkip` with either `verPass` or `verFail`.
- `Column 4` contains the time-out limit in number of seconds.
- `Column 5` gives the runtime in number of seconds.
- `Columns 6 through` $n-1$, where n is the total number of columns, can contain arbitary numerical data, such as number of implications, with appropriate headers.
- `Column` n contains a bit-string representation of the solution when the instance is satisfiable, or is empty if not.

To help with the gathering and viewing of statistical data, there are other files created from the tabular output during post-processing. These include *solverID*`.stat`, a table with statistical summaries of numerical data in columns 5 through $n-1$ of *solverID*`.tab`, *solverID*`_tab.html` and *solverID*`_stat.html`, html versions of the respective tables. Additional files that facilitate the creation of charts showing the *solvability function* (see [BSL03]) are also included.

STATISTICS. The primary purpose of the statistics stage is to allow the user to produce detailed tables useful for the creation of charts for measures of merit other than runtime. For example,

```
STATISTICS = moreStats
moreStats   = tab2stats_SAT.tcl I_implications,J_decisions,K_backtracks
```

specifies that the statistic-creation program that is normally applied to runtime (in column 5) only will also be used to analyze data columns with headers `I_implications`, `J_decisions`, and `K_backtracks`, respectively. In principle this stage could be used to run any program that analyzes or collects data in one directory of results at a time.

VIEWER. The only viewer script supplied with this release is `stat2latex.tcl`, which gathers data from statistics (`.stat`) files for various classes and solvers (specified by the variables `VIEW_SOLVERS` and `VIEW_CLASSES`) and creates a set of compilable LaTeX tables from them. Global variables at the beginning of the script control exactly which statistics are gathered and for which measures of merit. They also specify the criteria to be used for grouping items into a single table (e.g. same class, same solver, etc.) and how the table should be formatted (what headers to use with which statistics and what `printf`-style format should be used for each table column). See [BLS03, BSL03, LSB03] for examples of the tables generated.

We expect that a later release will allow users to specify many of these details in either a user-friendly configuration script or even a GUI (not necessarily for creating LaTeX tables, but for html files or charts).

4 Utility Programs

The following scripts and programs are supplied with `SATbed` in addition to the main controlling script (`@SATbed`):

- Sources for the solvers. These are in subdirectories of the `src` directory.
- An encapsulation script *solverID*`_encap.tcl` and a post-processing script *solverID*`_postProcess.tcl` for each supplied solver *solverID*.
- Scripts `sched_classic.tcl` and `sched_new_v04.tcl`, which can be used to generate the original and new scheduling instances discussed in [BSL03], respectively.
- Sources for the programs `cnfTrim` and `cnf3sat`. These programs modify formulae in DIMACS format to "trim" unit clauses (i.e. removethem by unit propagation) and convert them to equivalent 3-sat formulae, respectively.
- Source for the program `rotate_formula`, which generates one random instance of one of our equivalence classes [BLS03, BSL03] (in `src`).
- Source for the program `verify`, which is used to verify whether a reported solution does indeed satisfy the instance for which it was reported.
- The script `tab2stats_SAT.tcl`, used to produce statistical files related to runtime during post-processsing, but also capable of producing them for other measures of merit.
- The script `stat2latex.tcl` for producing LaTeX tables from statistical summaries during the `VIEWER` stage.

5 Configuration Files for Demonstration

The directory `demos` contains files `demo-1.cfg` through `demo-7.cfg`. Each of these configuration files has all the information needed to run all six stages on a simple example, but each of `demo-1.cfg` through `demo-6.cfg` is designed to run exactly one stage, while `demo-7.cfg` runs all stages. More specifically, the demo configuration files are all identical except for the setting of the `START` and `STOP` identifiers: for example, the file `demo-1.cfg` runs the `REFGEN` stage only and has `START = REFGEN` and `STOP = REFGEN`, the file `demo-2.cfg` runs the `CLASSGEN` stage only and has `START = CLASSGEN` and `STOP = CLASSGEN`, and so on. In `demo-7.cfg`, the settings are `START = REFGEN` and `STOP = VIEWER`.

References

[BLS03] F. Brglez, X. Y. Li, and M. Stallmann. On SAT Instance Classes and a Method for Reliable Performance Experiments with SAT Solvers. *Annals of Mathematics and Artificial Intelligence (AMAI), Special Issue on Satisfiability Testing*, 2003. Under review. Submitted to AMAI as the revision of the paper published at the Fifth International Symposium on the Theory and Applications of Satisfiability Testing, Cincinnati, Ohio, USA, May 2002. For a reprint, see `www.cbl.ncsu.edu/publications/` .

[BSL03] F. Brglez, M. F. Stallmann, and X. Y. Li. SATbed: An Environment For Reliable Performance Experiments with SAT Instance Classes and Algorithms. In *Proceedings of SAT 2003, Sixth International Symposium on the Theory and Applications of Satisfiability Testing, May 5-8 2003, S. Margherita Ligure - Portofino, Italy* , May 2003. A revised version available from `http://www.cbl.ncsu.edu/-publications/`. See also `http://www.mrg.dist.unige.it/events/-sat03/` and `http://www.cbl.ncsu.edu/OpenExperiments/SAT/` .

[HK01] E. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination, 2001. PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg, 2001.

[HK03] Edward A. Hirsch and Arist Kojevnikov. Unitwalk home page: Local search guided by unit clause elimination, 2003. available at `http://logic.pdmi.ras.ru/~arist/UnitWalk/`.

[LeB03] Daniel LeBerre. Opensat, 2003. Contact author — further information at URL `http://www.cril.univ-artois.fr/~leberre/recherche.php`.

[LSB03] X. Y. Li, M. F. Stallmann, and F. Brglez. QingTing: A Local Search SAT Solver Using an Effective Switching Strategy and an Efficient Unit Propagation. *Springer-Verlag Lecture Notes in Artificial Intelligence (LNAI), Special Issue on Satisfiability Testing*, 2003. This is a revised version of the SAT'2003 paper to be published in SAT'2003 LNAI issue by Springer-Verlag. Available at `www.cbl.ncsu.edu/publications/` .

[MMZ+01] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *IEEE/ACM Design Automation Conference (DAC)*, 2001. Version 1.0 of Chaff is available from `www.ee.princeton.edu/ chaff/zchaff/-zchaff.2001.2.17.src.tar.gz`.

[SK02] Bart Selman and Henry Kautz. WalkSAT Homepage: Stochastic Local Search for Satisfiability, 2002. The source code is available at `www.cs.washington.edu/homes/kautz/walksat/`.

[Zha97] Hantao Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, pages 272–275, 1997. Version 3.2 of SATO is available from `ftp://cs.uiowa.edu/pub/hzhang/sato/sato.tar.gz`.