SBSAT Quick-Start Guide

Michal Kouril, Sean Weaver, Andy Vogel

12th September 2003

SBSAT Version 2.0, August 2003

Contents

Wecolme to SBSAT (State Based Satisfiability) Solver

SBSAT is a software package for solving instances of a generalization of the well-known Satisfiability problem. In particular, the problem solved by SBSAT is the following:

```
Given:  Input variable set V={v1...vn} of Boolean vari-
ables
        Set of Boolean func-
tions F={f1...fm} where for all
        i, fi maps an assignment of values to vari-
ables of
        V to {T,F} (denote by V|t an assign-
ment t of values)
Result: An assignment t of values to variables of V such
        that for all i fi(V|t) = T, or "unsatisfi-
able" if
        no such assignment is possible.
```

If, for all i, fi is a function corresponding to the conjunction of a subset of variables of V, then the problem is reduced to the well studied Boolean Satisfiability Problem. If the variables of V are allowed to take arbitrarily many values, then the problem becomes the well-studied Constraint Satisfaction Problem.

The functions F may be specified in several different ways. There is one canonical input specification format, in terms of BDDs, which may be the target of any user-supplied translation from a particular user-defined set of functions. Specific supported input formats and examples of how to translate a user-defined format for use by SBSAT are given in Section˜\ref{the-examples-section}.

SBSAT executes in several phases. First, an instance is read from file. Second, depending on switches set on the command line when invoking SBSAT, various levels of preprocessing are applied to the input instance with the intention of producing an internal Satisfiability-equivalent set of constraints which supports smaller searches through judicious use of search heuristics and learning (Lemmas - see below). Details of the kinds of preprocessing available and their effects are given in Section˜\ref{details-section} and examples of their use are given in Section˜\ref{examples-section}. Third, the internal form is searched for a solution.

The user is given a choice of four ways to perform a search. These are:

1. Backtracking
2. Backtracking with Lemmas
3. BDD Walksat (incomplete solver)
4. WVF (I do not understand what your explanation is trying to say)

Search heuristics are used to help control the size of the search space. There are several provided:

1. Chaff-like
2. Johnson generatization
3. Others?

The size of the search space can be further controlled through learning. As backtracks occur, new constraints, called Lemmas, are added to the internal constraint set. These can prevent some fruitless backtracking later in the search.

Available input formats are...

1. CNF
2. BDD (ITE)
3. Truth Table (Smurf)
4. Tracer
5. XOR

```
read input --> preprocessing --> write output
                              \-> brancher   --> print solutions
                              \-> bddwalksat /
                              \-> wvf brancher
```

The Solver does not have GUI based interface, rather it uses a command line interface.

The solver was successfully tested and compiled on a number of Unix based platforms such as Linux, DEC, Solaris, Mac OS X, Windows/Cygwin with a number of different compilers such as gcc2.95, gcc3.x, solaris-cc, dec-cc, pgcc. (intel cc?).

1 About the Quick-Start Guide

The main purpose of the Quick-Start Guide is to introduce a new user to the SBSAT Solver. We will use a few well chosen examples to demonstrate the different functionality of the SBSAT solver without getting bogged down in details. For a more detailed description of the solver please see the SBSAT Manual[1].

All problems in this manual are part of the SBSAT distribution in the examples subdirectory.

1.1 Conventions

Program input and output

Fixed sized length font (like a typewriter) shows a line of text as it should appear on the computer screen.

(Example: `Reading file ...`)

Command line demonstration

A string character ($) at the beginning of a line represents the command-line prompt.

(Example: `$ ite small.cnf`)

Program options

Programming options appear in *Italics* to contrast with the option parameters, which appear in plain text.

(Example: `$ ite --help`)

Keywords in an input format

An input format example has keywords in **Bold**.

(Example: ***add_state** $1,2)

1.2 Hardware requirements

A Unix style operating system with c++ compiler installed is required. All examples require at least 32MB of RAM beyond the requirements of the operating system. Disk requirements depend on the operating system at least 200MB.

[1] not available yet

In general, SBSAT allocates as much memory as it needs. The amount of memory can be limited only indirectly by changing e.g. the number of lemmas it maintains in the cache or the size of the pools for different stacks (it allocates new pool of the same size if it runs out of the current one). There is no other option to limit the amount of memory it allocates. It is expected that the amount of available memory matches the size of the problem being solved. SBSAT is not multi-threaded and does not take advantage of multiple processors.

1.3 Definitions

- Preprocessing
- Backjumping, backjumps
- Lemma
- DIMACS CNF
- Solution
- Satisfiable output
- Unsatisfiable output
- Standard input and output
- for all, there exists
- imp, or, and, nor, nand, not, nimp, xor, equ, ...
- add_state
- '*' (star in the begining of the line)

1.4 How to compile

1.5 How to install

2 Running SBSAT: the basics

I see nothing that says where ite is to be run from, whether there needs to be a path variable set, what the directory structure is expected to be, or even where the example "small.cnf" is. A novice user will certainly give up right here! I would.

2.1 Simple CNF Example

Page 5: There is no such thing as a "CNF file". There could be a file whose contents represent an instance of CNF in, say, DIMACS format. Please always consider whether a term you are using has been defined somewhere previously. You might use CNF file as shorthand for something else but a novice user has no idea this is the case if not told.

We will start by using SBSAT to solve a simple CNF file. According to the DIMACS standards every CNF file starts with the header "`p cnf num_of_vars num_of_fns`" where `num_of_vars` is the number of variables present in the input file and `num_of_fns` is the number of functions present in the input file. Lines starting with `'c'` indicate a comment and are ignored. Functions are expressed as a string of positive and negative integers, representing clauses in conjunctive normal form. Each function (clause?) ends with `'0'` (number zero) and a '-' (minus) in front of a variable indicates negation.

Example (file small.cnf):

```
p cnf 6 8
c This is a demonstration of the CNF format for the SB-
SAT solver
1 2 3 0
2 3 4 0
3 4 5 0
4 5 6 0
-1 -2 -3 0
-2 -3 -4 0
-3 -4 -5 0
-4 -5 -6 0
```

To run the solver on this file, start the solver (`ite`) with the filename of this problem on the command line:

```
$ ite small.cnf
```

The output is as follows:

```
warning: ini file not found /home/fett/ite.ini
Reading File ../examples/small.cnf  ....
Reading CNF ... Done
Preprocessing .... Done
Creating Smurfs ... Done
Solution verified.
Time in brancher:  0.000 secs.
0.000 backtracks per sec.
Time: 0.000s. Backtracks: 0 (10000000.000 per sec) Progress:  0.00%
 Choices (total, dependent, backjumped): (3, 0, 0)
 Lemmas (cached, non-cached, added): (0, 2, 2)
 Inferences by smurfs: 2; lemmas: 0;
 Backtracks by smurfs: 0; lemmas: 0;
 Backjumps: 0 (avg bj len: 0.0)
Choice Points: 2 , Backtracks: 0, Backjumps: 0
Satisfiable
Total Time: 0.010
```

Page 6: What does "decoding the output: satisfiable" mean?

Decoding the output: `Satisfiable`

In order to get the actual satisfiable assignment from the solver we need to add the input parameter instructing the solver to output the solution.

```
$ ite -R r small.cnf
```

TIP The order of the parameters on the command line does not matter. (With the exception of *-All* preprocessing switch and preprocessing enable/disable switches). So in this case the following command line would do exactly the same as the one above.

```
$ ite small.cnf -R r
```

Output:

```
warning: ini file not found /home/fett/ite.ini
Reading File ../examples/small.cnf  ....
Reading CNF ... Done
Preprocessing .... Done
Creating Smurfs ... Done
Solution verified.
Time in brancher:  0.000 secs.
0.000 backtracks per sec.
```

```
Time: 0.010s. Backtracks: 0 (1000000.000 per sec) Progress:  0.00%
 Choices (total, dependent, backjumped): (3, 0, 0)
 Lemmas (cached, non-cached, added): (0, 2, 2)
 Inferences by smurfs: 2; lemmas: 0;
 Backtracks by smurfs: 0; lemmas: 0;
 Backjumps: 0 (avg bj len: 0.0)
Choice Points: 2 , Backtracks: 0, Backjumps: 0
// Solution #1
Preprocessing .... Done
-1 -2 3 4 -5 -6
Satisfiable
Total Time: 0.020
```

The default output mixes solution information with execution information. Solution information may be separated from execution information as follows.

```
$ ite small.cnf -R r --output-file output.txt
```

output.txt:

```
// Solution #1
-1 -2 3 4 -5 -6
```

> Page 7: There is no explantion given of the meaning of the solution information. Add some.

TIP Some of the command line options have both a short and a long flag. They can be used interchangably. For example the '*-R*' option is also '*--show-result*'.

Note: dash (-) instead of the filename denotes the standard input or standard output depending on the context.

> Page 7: It says dash denotes standard input. Now what? No example?

All available options can be printed by specifying '*--help*'.

```
$ ite --help
```

Shortened output:

```
SBSat is a SAT solver. Usage:
ite [OPTIONS]... [inputfile] [outputfile]
Options:
--help, -h              Show all program options
--version               Show program version
--create-ini            Create ini file
--ini <string>          Set the ini file [default="~/ite.ini"]
```

```
--debug <number>       debugging level (0-none, 9-max) [default=2]
--debug-dev <string>   debugging device [default="stderr"]
--params-dump, -D      dump all internal parameters before processing
--input-file <string>  input filename [default="-"]
--output-file <string> output filename [default="-"]
--temp-dir <string>    directory for temporary files [default="$TEMP"]
--show-result <string>, -R <string>
                       Show result (n=no result, r=raw, f=fancy)
                       [default="n"]
...
```

2.2 Converting your problem into a SAT problem

Page 7: The section on converting to SAT does not belong here. At this point the user is trying to figure out how to use SBSAT. Either move it to a later section, probably on its own, or save it for the detailed sections (I still believe we will have one user manual when we are done and one documentation manual).

Page 8: Conventions section should include all conventions you use. Thus, "for all" and "there exists" and so on are in the conventions section.

Page 8: What does "*imp(1,2)" mean? Conventions again, maybe?

Any NP-Complete problem can be translated into a SAT problem. Two such problems are demonstrated here. The first is a problem from AI where we have a database of information and we'd like to ask a question about the database. Here is a sample database taken from Charles Dodgson's (aka Lewis Carroll) book *Symbolic Logic*:

1. Coloured flowers are always scented.
2. I dislike flowers that are not grown in the open air.
3. No flowers grown in the open air are colourless.

We would like to ask this question - Do I dislike all flowers that are not scented? Now we must translate these statements into First Order Logic. This gives:

1. $\forall w$ (Colored(w) $\Rightarrow$ Scented(w))
2. $\forall x$ ($\neg$OpenAir(x)$\Rightarrow$ $\neg$Like(x))
3. $\forall y$ (OpenAir(y) $\Rightarrow$ Colored(y))

The question becomes: $\forall z$ ($\neg$Scented(z) $\Rightarrow \neg$Like(z))

We can now translate these statements into one of the acceptable formats used by SBSAT. Both CNF and ITE(BDD) are demonstrated here.

Example(file flowers.ite):

```
p bdd 4 4
;Coloured flowers are always scented.
;I dislike flowers that are not grown in the open air.
;No flowers grown in the open air are colourless.
;Do I dislike all flowers that are not scented?
;Where Col-
ored=1, Scented=2, Like=3, Grown in the Open Air=4
;The question must be negated and added to the database.
;If SBSAT returns 'unsat' then the answer to the ques-
tion
;is 'YES', otherwise the answer is 'NO'.
*imp(1, 2)
*imp(-3, -4)
*imp(3, 1)
*not(imp(-2, -4))
```

Example(file flowers.cnf):

```
p cnf 4 5
c Coloured flowers are always scented.
c I dislike flowers that are not grown in the open air.
c No flowers grown in the open air are colourless.
c Do I dislike all flowers that are not scented?
c Where Col-
ored=1, Scented=2, Like=3, Grown in the Open Air=4
c The ques-
tion must be negated and added to the database.
c If SBSAT returns 'unsat' then the answer to the ques-
tion
c is 'YES', otherwise the answer is 'NO'.
-1 2 0
3 -4 0
-3 1 0
```

```
-2 0
4 0
```

2.3 Choosing a different solver

All pages: is -bw the same as -b -w or what? (Some people might assume it is).

Solver	Default	Option	Description
Smurf Brancher	yes	*-b*	
BDD WalkSAT	no	*-w*	
WVF Brancher(obsolete)	no	*-m*	

2.4 Converting the input file

Section 2.4: I have no idea what you are talking about.

-In 0 -All 0 is the only way to get a direct translation of the original file (aka, no preprocessing whatsoever).

2.5 BDD (ITE) format examples

2.5.1 Structure of BDD(ITE) format

Page 9: You wait to here to tell me what the "*" is? We already used it on a previous page!!! Conventions maybe?

Vicinity of Page 9. Somewhere we have to be more specific about limitations. The user is led to believe that putting "*" in front of anything will give a function that SBSAT can create. I do not think this is the case and the user must be informed of some rules that will prevent a crash.

As with the CNF format the file starts with the header '`p bdd num_inp_vars num_fns`'

Each line starting with start '`*`' denotes a new BDD function. The table shows the basic built-in functions:

Function	Number of params		
equ	2+		
and	2+		
or	2+		
and	2+		
not	1		
imp	2+		
ite	3		
xor	2+		

The parameters of the functions are either variables in the form of numbers or another function in the form of `'$'` and the index of the function in the file (starting with 1).

For the functions where the number of parameters can vary the number of parameters is attached right after the function name. Example:

```
xor3(1, 2, 3)
```

2.5.2 Simple XOR Example

Here is the file called xortest.ite:

```
p bdd 7 3
*equ(xor( xor(1, 2), 3), F)
*equ(xor(3, and3(5, 4, 6)), T)
*equ(xor(and3(1, 2, 3), and3(4, 5, 7)), F)
```

Run the example:

```
$ ite xortest.ite
```

The output:

```
warning: ini file not found /home/fett/ite.ini
Reading File ../examples/xortest.ite  ....
Reading ITE ... Done
Preprocessing .... Done
Creating Smurfs ... Done
Solution verified.
Time in brancher:  0.000 secs.
0.000 backtracks per sec.
Time: 0.000s. Backtracks: 0 (10000000.000 per sec) Progress:  0.00%
  Choices (total, dependent, backjumped): (3, 0, 0)
  Lemmas (cached, non-cached, added): (0, 5, 5)
  Inferences by smurfs: 5; lemmas: 0;
  Backtracks by smurfs: 0; lemmas: 0;
  Backjumps: 0 (avg bj len: 0.0)
Choice Points: 2 , Backtracks: 0, Backjumps: 0
Satisfiable
Total Time: 0.020
```

2.5.3 add_state example

Section 2.5.3: add_state is a really bad name because the user may easily get confused about the meaning of state. Perhaps add_duplicate_function is more like what the user is expecting. The section also needs to explain why we have such a crazy operation anyway.

Another important function is **add_state**. This function has two parameters. The first one is a BDD function. The second parameter is the shift in the variable number. **Add_state** creates a new BDD with structure identical to the first parameter but with each variable incremented by the second parameter.

```
p bdd 44 5
*equ(xor(1, and(-17, 33)), ite(15, or(33, -40), -33)))
*add_state($1, 1)
*add_state($1, 2)
*add_state($1, 3)
*add_state($1, 4)
```

The example will be expanded into the following form:

```
p bdd 44 5
*equ(xor(1, and(-17, 33)), ite(15, or(33, -40), -33)))
*equ(xor(2, and(-18, 34)), ite(16, or(34, -41), -34)))
*equ(xor(3, and(-19, 35)), ite(17, or(35, -42), -35)))
*equ(xor(4, and(-20, 36)), ite(18, or(36, -43), -36)))
*equ(xor(5, and(-21, 37)), ite(19, or(37, -44), -37)))
```

2.5.4 print_tree and pprint_tree examples

2.5.5 Function definition example

```
#define g 1 2 3 # ite(1, 2, and(-2, 3))
```

2.5.6 Complex example

```
p bdd 18 13 ; 18 vars, 13 functions
#define fun 1 2 3 4 # ite 1 and 2 3 or 3 4 ; parms must be consecutive inte-
gers from 1
#define g 1 2 3 # ite(fun(1, 2, 3, -2), T, 3)
InitialBranch (2, 4..12, 15, 16, 18, 1, 3) ; These vari-
ables will be branched on first
ite 4 5 6 ; eqn $1
*or $1 3 ; eqn $2, smurf 1
*or 5 -6 ; eqn $3, smurf 2
*and $1 -4 ; eqn $4, smurf 3
*imp $1 $4 ; eqn $5, smurf 4
```

```
#define imp 1 2 # or3(1, 2, 3); Notice the 'imp' operator was overloaded.
*imp(3 4 5)    ; eqn $6, smurf 5
               ; this is a really bad idea, it probably shouldn't be allowed
print_tree $5  ; no equation created, no smurf created
pprint_tree $5 ; no equation created, no smurf created
ite(2,              ; eqn $7, no smurf created
      ite(3, 4, 5), ; plus comments are ignored, even in the middle of a func-
tion
      ite(4, 5, F))
*fun 4 -5 2 3 ; eqn $8, smurf 6
*fun g -5 4 $6 2 3 4 ; eqn $9, smurf 7
*equ(5, xor3(and(-3, 4), nand(7, 5), ite(15, 4, nor(4, -
7)))) ; eqn $10, smurf 8
*add_state($10, 1) ; eqn $11, smurf 9
                   ; add_state creates a BDD which is identical to the first
                   ; argument but with all it's variables incremented by the
                   ; second argument.
*add_state($10, 2) ; eqn $12, smurf 10
*add_state($10, 3) ; eqn $13, smurf 11
```

TIP Although both parentheses and commas are optional, we recommend you keep them in so you can easily orient yourself in the file.

3 SBSAT Advanced

3.1 Changing Preprocessor Options

Page 13: There really needs to be some explanation of the preprocessing options. Also some tips about when a preprocessing option might pay off and when might it be a liability.

The available preprocessing options are :

Name	Default	Short	Formats	Description
Clustering	yes	Cl	CNF	
Cofactoring	yes	Co	All	
Pruning	yes	Pr	All	
Strengthening	yes	St	All	
Inferences	yes	In	All	
Existential Quantification	yes	Ex	All	
Dependent Var. Clustering	yes	Dc	All	

Preprocessing sequence: `(ExDc)*(ExSt)*(ExPr)*`

The sequence in which the preprocessing options are applied is specified by the 'preprocessing sequence' string. The parentheses '()' border the repeated sequences and are followed with the number of repeats. A star '*' means repeat until there is no change.

Example:

```
$ ite --preprocess-sequence '(ExDc)3(ExSt)2(ExPr)10' small.cnf
```

(or `$ ite -P '(ExDc)3(ExSt)2(ExPr)10 small.cnf'`)

For some problems the preprocessing might take too long or may not produce a desired effect. Therefore it is possible to enable or disable the preprocessing options or change their sequence.

Example:

```
$ ite -St 0 small.cnf
```

TIP How to avoid repeating long preprocessing: Save the problem after preprocessing in SMURF file format (Using `$ ite --output-file newfile.smurf -s myoldfile`) and disable the preprocessing next time you run your problem (Using `$ ite --All 0 newfile.smurf`) .

3.2 Changing heuristic and its options

Page 13: We do not implement a Johnson heuristic. We implement a heuristic that is designed to skew locally by favoring inferences and balance globally by favoring relatively equal "probabilities" of satisfying search subtrees. Call this the LSGB for Locally Skewed, Globally Balanced heuristic. Say that LSGB becomes Johnson when F is a bunch of disjunctions and k=2. Mention that the parameter k controls how much inferences are favored (deep vs. shallow) and can have a major impact on performance. Mention when high k might pay off, same for low k.

The standard brancher has to two available heuristics, the Johnson heuristic and Lemma (chaff-like) heuristic. Both heuristics have their advantages and disadvantages. Usually if one heuristic is better than the other on a specific problem then this is also the case for similarly structured problems.

Example:

Heuristic	Default	Option	Description
Johnson heuristic	yes	`-H j`	
Lemma heuristc	no	`-H l`	
Combined heuristic	no	`-H jl`	

TIP Using the Combined heuristic (`-H jl`) might be a good compromise if you are unsure which heuristic to choose.

3.3 Adjusting other parameters

Page 14: Say what a lemma is, say that lemmas are stored with a particular replacement policy. Say what a backjump is. Conventions? Perhaps change Conventions section to Conventions and Definitions and stick in some definitions that you are right now assuming that everyone knows about.

3.3.1 Lemma cache

The size of the cache in which the lemmas are stored is fixed throughout the branching process. Necessary amount of memory is automatically allocated to accomodate all lemmas in the cache. Usually the bigger the memory cache

the slower the branching process. Therefore increasing lemma cache might not improve the overall branching time.

The parameter to use for controlling the lemma cache is `-L` or `--max-cached-lemmas`.

Example:

$ ite -L 1000 problem.cnf

It is possible to set the lemma cache to 0. This will prevent any lemma from being stored beyond the point they are needed. The brancher will still generate lemmas during backtracking and inferencing.

To prevent the lemmas from being created and used the backjumping must be disabled together with setting the lemma cache to 0.

Example:

$ ite -L 0 –backjumping 0 slider_80_unsat.ite

For some problems this will yield significantly better results than when the lemmas are used.

Note: the lemma heuristic is not compatible with the situation when the lemmas are disabled. Also the effectivness of the lemma heuristic is decreasing with descreasing the lemma cache.

> Page 15: Now you say what backjumps are???!!! Stick in the new Conventions and Definitions section.

Backjumping refers to the backtracking property where the literal previously chosen by heuristic and not involved in contradition is not considered with the reversed sign.

3.3.2 Controlling the time on the preprocessing and brancher

> Section 3.3.2: what does it mean for the preprocessing to take too long? You might want to say that preprocessing time exceeds the savings in time realized by the search engine, if that is what you mean.

In some situations the preprocessing takes too long. One way to interrupt the preprocessing is to change the preprocessing string to perform less iterations through the preprocessing options (see ...). Another way to cut the preprocessing time is to specify the time limit in seconds for how long the preprocessing

can take. After the time limit has been reached the preprocessor will quit and the control will be handed to the brancher.

Example:

```
£ ite small.cnf --max-branching-time 180
```

This will allow 3 minutes for preprocessing and continue to the brancher after that.

> Section 3.3.2: If the preprocessor is cancelled after some time limit, what is guaranteed about the resulting massaged input that is handed to the search engine, if anything?

The similar option exists for the brancher:

```
$ ite small.cnf --max-preproc-time 180
```

3.3.3 Creating and using an ini file

> Section 3.3.3: This section should be higher in the section hierarchy. (You don't need an ini file to run sbsat)

If you are working on a problem that requires adding the same command line options over and over it is better to create an ini file. SBSAT automatically looks for ite.ini in your home dicrectory.

To create an ini file with the default values for all available options

```
$ ite --create-ini > ~/ite.ini
```

You can edit the ini file and change the values to those of your choice. Please note that the command line options take precedence before the ini file settings. This way you can effectively override all settings.

Also you can create different ini files for different problems. To use them use –ini option. Example:

```
$ ite --ini myini.ini small.cnf
```

TIP You can specify the name of the input file in the ini file (using `input-file="small.cnf"`) and start sbsat with the ini file only (using `$ ite --ini myini.ini`).

3.4 Debugging your problem

- try converting to another format
- debug prints (in ITE format)

- print internal data from the solver:
- be familiar with BDDs
- output the BDDs before preprocessing
- match them to your original problem
- if you think you discovered a bug in SBSAT email us!

3.5 Troubleshooting the compilation

- use different compiler

```
$ ./configure CXX=g++
```

- link the libraries staticly

```
$ ./configure --static
```

4 Getting more help

- Read the SBSAT Manual[2]
- Check out the SBSAT Web Pages[3]
- Email us:
  John Franco franco@gauss.ececs.uc.edu
  Michal Kouril mkouril@ececs.uc.edu
  Sean Weaver fett@gauss.ececs.uc.edu

Vicinity of Page 8: Man are things out of order. Let's fix the sections first then we can talk about section order.

I can see sections 2.5.4, 2.5.5, and 2.5.6 are not ready yet.

Sections 3.4 and 3.5 are not ready, I see.

Why is the order of some charts ... default, option ... and some ... option, default... This is really confusing!! Can we have some standardization?

Page 14: lemmas are disable => lemmas are disabled

Section 3.3.2: change the title to "Controlling the time on..."

[2] not available yet
[3] not available yet