Getting Started With SBSAT

Michal Kouril, Sean Weaver, Andy Vogel

31st July 2003

SBSAT Version 2.0, August 2003

Contents

Wecolme to SBSAT (State Based Satisfiability) Solver

SBSAT is Unix based SAT (Satisfiability) solver for ... The solving begins with the user encoding the input problem into one of the input formats acceptable by SBSAT into a single file. The SBSAT solver first reads the input, runs the preprocessor which massages and prepares the input for the brancher. The preprocessed problem goes either into one of the branchers or is converted into a different format.

Available branchers:

- Backtracking Brancher
- Backtracking Brancher with lemmas
- BDDWalkSAT - Incomplete Solver
- WVF Brancher Original brancher without lemmas and only one heuristic

Available formats BDD (ITE), CNF, SMURF, TRACE

Picture here:

```
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?).

The SBSAT Solver is free under the GPL licence.

1 About this manual

The main purpose of this manual 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 options

Italic (Example: `$ ite --help`)

Directive in each input formats

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

Command line demonstration

Start with the string character ($)

(Example: `$ ite small.cnf`)

Command line intput and output of the program

Fixed sized length font (Example: `Reading file ...`)

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.

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] not available yet

2 SBSAT Basics

2.1 Simple CNF Example

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
```

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
```

You can see that the solution is mixed with the normal output of the solver. Therefore it might be better to specify the output file for the solution.

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

output.txt:

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

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.

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

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

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

2.4 Converting the input file

-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

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

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

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

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

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

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 sliders_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 disable. Also the effectivness of the lemma heuristic is decreasing with descreasing the lemma cache.

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 Constraining the time on the preprocessing and brancher

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.

The similar option exists for the brancher:

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

3.3.3 Creating and using an ini file

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 real 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

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