Getting Started With SBSAT

Michal Kouril

25th July 2003

SBSAT Version 2.0, August 2003

Contents

Wecolme to SBSAT (abrev?) 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 preprocessor which .... Preprocessed problem goes either into one of the branchers or is converted into a different format:

Available branchers Classic Brancher Backtracking brancher with lemmas BDDWalkSAT Non— 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 – it is rather use command line interface.

The solver was successfully tested on 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 GPL licence.

1 About this manual

The main purpose of this manual is to introduce the user to the SBSAT Solver. We will use a few well chosen examples to demonstrate different functionality of the SBSAT solver without getting bogged down in details. For the more detailed description of the solver please see the SBSAT Manual*.

All problems in this manual are part of the SBSAT distribution in the tests 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: `blah blah`)

1.2 Hardware requirements

Unix style operating system with c++ compiler installed. All examples require at least 32MB?? of RAM beyond the requirements of the operating system. Disk requirements

Limits In general, SBSAT allocate as much memory as it needs. There is not 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 any advantage of multiple processors.

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_var num_of_fn`" where `num_of_var` is the number of variables present in the input file and `num_of_fn` is the number of functions present in the input file. Lines starting with `'c'` indicate comment and are ignored. Functions are in conjuctional normal form, each function ends with `'0'` (number zero). '-' (minus) in front of a variable indicates negation.

Example (file small.cnf):

```
p cnf 6 8
c This is the demonstration of the CNF for-
mat c for the SBSAT 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:

```
.... (copy) ...
```

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 like 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:

....

You can see that the solution is mixed with 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:

....

TIP Some of the command line options have to versions. Long and short. They can be used interchangably. For example '*-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 could be printed by specifying '*--help*'.

```
$ ite --help
```

Shortened output:

```
SBSat is 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 SAT problem

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

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_fn`'

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		

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 1000 1000
*equ(xor( xor(123, 125), 156), f)
*equ(xor(134, and3(155, 127, 167)), t)
*equ(xor(and3(1, 2, 3), and3(45, 145, 167)), f)
```

Run the example:

```
$ ite xortest.ite
```

The output:

... (c&p) ...

2.5.3 add_state example

Another important function is add_state. This function has two parameters. First one is the BDD function specified as the $ and the one-based index of the function in the file. The second parameter is the shift in the variable number.

```
p bdd 80 83
*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 80 83
*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 Function definition example

```
#define g 1 2 3 # ite f 1 2 3 -2 T
```

2.5.5 Complex example

```
InitialBranch (2, 4..12, 15, 16, 18, 1, 3) ;
; These variables will be labeled as  independent variables,
; the rest are labeled as dependent
print_tree $5; no equation created, no smurf created
pprint_tree $5 ; no equation created, no smurf created
```

<u>TIP</u> Although neither parentheses nor commas are required (you can leave them out) we recommend you keep them 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	All	
Cofactoring	yes	Co	CNF	
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 is reapeats. Star '*' means repeat until there is not change. You can replace the star with a number.

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 without the desired effect. Therefore it is possible to change the preprocessing options turn them off/on 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 `--output-file` and `-s`) and disable (`--All 0`) the preprocessing next time you run your problem.

3.2 Changing Heuristic and its options

Standard brancher has to two available heuristic schemas. Johnson heuristic and Lemma (chaff-like) heuristic. Both heuristic have their advantages and disadvantages. Usually if one heuristic is better than the other on a specific problem then it is the case for the similar problems.

Example:

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

TIP Using the Johnson 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 (non lemma brancher)

3.3.2 Constraining the time on the preprocessing and brancher

In some situations the preprocessing time takes too long. One wayt 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 how long the preprocessing can take. After the time is up preprocessor will quit and the control handed to the brancher.

Example:

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

This will allow 3 minutes for preprocessing and quits after that.

The similar option exists for the brancher:

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

3.3.3 Creating and using 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 the ini file starter with the default values for all available options

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

You can edit ite.ini and change the default 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.

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 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[1]
- Check out the SBSAT Web Pages[2]
- Email us:
  John Franco franco@gauss.ececs.uc.edu
  Michal Kouril mkouril@ececs.uc.edu
  Sean Weaver fett@gauss.ececs.uc.edu

[1] not available yet
[2] not available yet