SBSat Manual

5th August 2003

Written by Michal Kouril, Sean Weaver,... <add your name here>
Covers the versions up to 2.0

Contents

1 Introduction

Introduction

2 Overview

Overview

3 Getting SBSat

SBSat is still considered research software. Please email franco@gauss.ececs.uc.edu for more information how to get it.

4 Compiling SBSat

Considerning you obtained sbsat as tar.gz archive you will need to unpack it first. Necessary utilities are tar and gzip. They are present on most platforms. Also most platforms will let you use the following command to do so:

```
# tar -xvzf sbsat.tar.gz
```

If this command fails please try

```
# gzip -dc sbsat.tar.gz | tar -xvf -
```

The result should be the directory with the sources. Change the current directory into newly created one:

```
# cd sbsat
```

And run configure and make:

```
# ./configure
# make
```

(optional) Using the following command you can run a few included benchmarks:

```
# make check
```

(optional) You may install the executable and associtated files onto your system using:

```
# make install
```

The advantage of this step is that typically the systems have the path where the executable is installed (/usr/local/bin) as part of the system PATH variable so you will not need to specify the full path to start the executable.

5 Running SBSat

SBSat executable is called ite and is located in the src directory within the directory containing the sources for sbsat (sbsat directory). Unless you installed it to your system in the previous step (make install) you will need to refer to it with the full path or add this path into your PATH variable. From now on I will assume that ite was installed.

The usage of SBSat is:

```
ite [options] [inputfile [outputfile]]
```

There are two basic options required by GNU standard:

```
# ite --version
```

This displays the current version.

```
# ite --help
```

Shows all the command line options.

If ite is started without parameters it expects the input data on standard input.

The first parameter without dash is the input data file, the second parameter without is the output file.

6 Program Options

6.1 General options

SBSat general options are:

–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"]
–verify-solution <number> Verify solution [default=1]
*–expected-result <string>*Report error if the result is not as specified
Options are SAT, UNSAT, TRIV_SAT, TRIV_UNSAT, SOLV_S
and SOLV_UNSAT [default=""]
–ctrl-c <number> Enable/Disable Ctrl-c handler to end preproc/brancher

[default=0]
–reports <*number*> Reporting style during branching (0 - standard, 1 - crtwin)
[default=0]

6.2 BDD table options

BDD options:
–num-buckets <*number*> Set the number of buckets in power of 2 [default=16]
–size-buckets <*number*> Set the size of a bucket in power of 2 [default=5]

6.3 Input options

Input options:
–limit-and-equ <*number*>
The minimum # of literals to flag sp. function and_eq
u
[default=2]
–limit-or-equ <*number*> The minimum # of literals to flag sp. function or_equ
[default=2]
–limit-or <*number*> The minimum # of literals to flag sp. function plaino
r
[default=8]
–limit-xor <number> The minimum # of literals to flag sp. function plainx
or
[default=5]
–break-xors <*number*> Break XORS into linear and non-linear functions
[default=1]

6.4 Output options

Output options:
-b Start regular brancher [default]
-w Start walksat brancher
-m Start WVF brancher
-n Don't start any brancher or conversion
-s Output in SMURF format
-c Output in CNF format
-p Output in tree like format
–cnf <*string*> Format of CNF output (3sat, qm, noqm)
[default="noqm"]
–tree, -tree Print tree representation
–tree-width <*number*>, *-W* <*number*> Set tree width [default=64]

6.5 Trace format options

Trace format options:

–module-dir <string> directory to find extra modules [default="./Modules"]

6.6 Preprocessing options

Preprocessing options:

–preprocess-sequence <string>, -P <string>
The preprocessing sequence
[default="(ExDc)*(ExSt)*(ExPr)*"]

–All, -All Enable/Disable All Preprocessing Options (1/0)

–Cl <number>, -Cl <number> Enable/Disable Clustering (1/0) [default=1]

–Co <number>, -Co <number> Enable/Disable Cofactoring (1/0) [default=1]

–Pr <number>, -Pr <number> Enable/Disable Pruning (1/0) [default=1]

–St <number>, -St <number> Enable/Disable Strengthening (1/0) [default=1]

–In <number>, -In <number> Enable/Disable Inferences (1/0) [default=1]

–Ex <number>, -Ex <number> Enable/Disable Existential Quantification (1/0) [default=1]

–Dc <number>, -Dc <number> Enable/Disable Dependent Variable Clustering (1/0)
[default=1]

–max-preproc-time <number> set the time limit in seconds (0=no limit)[default=0]

6.7 Brancher options

Brancher options:

–reverse-depend, -r Reverse dependency info on in/dependent variables

–clear-depend, -e Clear dependency information on variables

–heuristic <string>, -H <string>
Choose heuristic j=Johnson, l=Chaff-like lemma h. i=Interactive [default="j"]

–backjumping <number> Enable/Disable backjumping (1/0) [default=1]

–max-cached-lemmas <number>, -L <number>
set the maximum # of lemmas [default=5000]

*–max-solutions <number>*Set the maximum number of solutions to search for.[default=1]

–sbj <number> Super backjumping. [default=0]

–max-vbles-per-smurf <number>, -S <number>
set the maximum number variables per smurf [default=8]

*–backtracks-per-report <number>*set the number of backtracks per report[default=10000]

*–max-brancher-time <number>*set the time limit in seconds (0=no limit)[default=0]

–max-brancher-cp <number> set the choice point limit (0=no limit) [default=0]

–brancher-trace-start <number>
number of backtracks to start the trace (when debug=9)[default=0]

–compress-smurfs <number> Share states among smurfs [default=1]
–smurfs-share-paths <number> Share paths among smurfs [default=0]

6.8 Johnson heuristic options

Johnson heuristic options:
–jheuristic-k <number>, -K <number> set the value of K [default=3.000000]
–jheuristic-k-true <number> set the value of True state [default=0.000000]
–jheuristic-k-inf <number> set the value of the inference multiplier [default=1.000000]

7 Input formats

The ITE Interface is the part of the program that reads in and interprets the user's input file. This process is as follows:
1. Format for Input Files

7.1 BDD (ITE) input format

The ITE interface is a continuation of a project started by Laura A. Pugh, August 17 2000 for C123 under the direction of Mark Vanfleet, Michael Dransfield, and Kelly McGuire. This project is being continued by Sean Weaver, under the direction of Mark Vanfleet and Dr. John Franco. The ITE interface is a module which can be used as a front end on any Satisfiability solver (SAT solver, see page ??).

The ITE interface provides the user with a unique Binary Decision Diagram (BDD) interface (See BDDs. page ??). This interface is used to enter problems into any SAT solver. BDDs allow the user to express his/her problem data in a way that is natural to it's domain. The common practice of translating a problem into Conjunctive Normal Form (CNF) destroys naturally occurring clusters of information that can be taken advantage of during the search process (See CNF, page ??). Receiving the input in BDD form also gives traditional SAT solvers the ability to add and use powerful BDD preprocessing techniques before search. During search a SAT solver also has the ability to use any information contained in the BDDs to help it's search heuristic. ITE can provide a translation to CNF if it is being used in conjunction with a CNF SAT solver. ITE is currently being used in conjunction with a SAT solver named SBSat. SBSat is being developed at the University of Cincinnati by Mark Vanfleet, Michael Dransfield, Dr. John Franco, Dr. John Schlipf, Sean Weaver, Michal Kouril, and George Vogel.

There is a header for every input file. Only comments are allowed to proceed the header. Thus it follows that all equations, defines, etc., follow directly after the header.

All equations are entered in prefix format using specific keywords and variables. There must be some sort of white space between all keywords and vari-

ables. White space is defined to be a blank, a tab, a carriage return, a comma, or a parentheses. In order to include comments in the input file, place a semi colon (";") prior to the comment. This will make the rest of the line a comment and ITE will not attempt to process it. Comments are not necessary for the program to operate, but are a convenience for the user.

7.1.1 File Header

In every input file, no equation can precede the header. The header conforms to the DIMACS standard which is as follows.

```
p bdd #Vars #Equations
```

The letter 'p' at the start of a line denotes the start of the header. ITE requires that 'bdd' be the next three letters; these denote that the file is in the BDD format. #InputVars is a number that and tells ITE how many input variables are going to appear in this input file. #Equations is a number that tells ITE how many equations appear in this input file. Here is an example of a valid header.

```
p bdd 56 97
```

This header tells ITE that the input file is in BDD format with 56 input variables and 97 equations.

7.1.2 Keywords

Keywords are not case sensitive. The current keywords used, with a brief description of each are as follows:

INITIALBRANCH

InitialBranch allows the user to specify which variables should be branched on first during search. InitialBranch is a command that should only be used when the user is sure he knows more information about the nature of the input file the attached solver will be able to derive. This keyword takes any number of positive integer arguments

```
Example: InitialBranch( 1, 3, 5..18, 4, 24..39)
This will mark variables 1, 3, 5..18 (inclu-
sive), 4, 24..39 (inclusive) to be branched on first.
*Note that this keyword is the only keyword which re-
quires parenthesis around it's argument list*
```

VAR

VAR is the base keyword to all equations. It takes one(1) argument, which must be an integer. VAR creates a BDD for the variable following it. This BDD

will have a true branch and a false branch.
The keyword VAR has been made obsolete. It is no longer necessary to place VAR in front of every variable. It is now assumed that all integers are variables unless preceded by a '$'. All integers preceded by a '$' refer to equations instead of variables.

NOT

The NOT function takes one(1) argument and forms a BDD which represents the negation of that argument. When negating a single variable, a negative sign ("-") can be placed directly in front of the variable as a shortcut (See Shortcuts)

ITE

The ITE function takes three(3) arguments and forms a BDD which expresses the If-Then-Else relation of the three(3) arguments. The function builds a BDD so that if the first parameter evaluates to true, then the second parameter must be true, else the third parameter must be true. Every BDD possible can be expressed using the ITE function.

```
Example: ITE( ITE( 3, $5, -5), 4, 7)
Assume BDD $5 has been previously defined.
*Note that the use of parenthesis and commas here is al-
lowed but not required*
```

AND

The AND function takes two(2) arguments and creates a BDD which expresses the logical AND of the two(2) arguments.

```
Example: AND 3 4
```

AND#

AND followed directly by a positive integer is a shortcut designed to allow the user to send more than two(2) arguments to the AND function. Any number of arguments is supported here.

```
Example 1: AND4(1, 2, 3, 4)
Example 2: AND5(-1,
                  AND3(2, 3, $4),
                                  5, $6, 7)
Assume $4 and $6 have been previously defined.
*Note that the use of white space here is al-
lowed but not required*
```

NAND

The NAND function takes two(2) arguments and creates a BDD which expresses NOT of the logical AND of the two(2) arguments.

NAND#

NAND followed directly by a positive integer is a shortcut designed to allow the user to send more than two(2) arguments to the NAND function. Any number of arguments is supported here.

OR

The OR function takes two(2) arguments and creates a BDD which expresses the logical OR of the two(2) arguments.

OR#

OR followed directly by a positive integer is a shortcut designed to allow the user to send more than two(2) arguments to the OR function. Any number of arguments is supported here.

NOR

The NOR function takes two(2) arguments and creates a BDD which expresses the NOT of the logical OR of the two(2) arguments.

NOR#

NOR followed directly by a positive integer is a shortcut designed to allow the user to send more than two(2) arguments to the NOR function. Any number of arguments is supported here.

EQU

The EQU function takes two(2) arguments and creates a BDD which expresses that these two(2) arguments are equal to each other.

EQU#

EQU followed directly by a positive integer is a shortcut designed to allow the user to send more than two(2) arguments to the EQU function. Any number of arguments is supported here.

XOR

The XOR function takes two(2) arguments and creates a BDD which expresses that these two(2) arguments are opposite of each other.

XOR#

XOR followed directly by a positive integer is a shortcut designed to allow the user to send more than two(2) arguments to the XOR function. Any number of arguments is supported here.

IMP

The IMP function takes two(2) arguments and creates a BDD which expresses the logical implication of the two(2) arguments.

Take note that IMP does not support multiple arguments as many of the other keywords do, this is because the implication operator has strict variable ordering dependence.

NIMP

The NIMP function takes two(2) arguments and creates a BDD which expresses the NOT of the logical implication of the two(2) arguments.

Take note that NIMP does not support multiple arguments as many of the other keywords do, this is because the implication operator has strict variable ordering dependence.

GCF

The GCF function takes two(2) arguments. This function finds all solutions of both parameters and then creates a BDD which represents the intersection of these solutions. GCF stands for Generalized Co-Factoring.

STRENGTHEN

The STRENGTHEN function takes two(2) arguments, e1 and e2. This function uses the information in e2 to strengthen e1. (See Strengthening, page ??)

RESTRICT

The RESTRICT function takes two(2) arguments, e1 and e2. This function prunes all branches found in e2 away from e1. (See Restrict, page ??)

EXIST

The EXIST function takes two(2) arguments, the first argument can be an equation or a variable (e1), the second argument must be a variable (v1). This function searches for and finds everyplace in e1 where v1 occurs. Every time v1 is found it is replaced by the logical OR of it's true branch and it's false branch. This function effectively removes v1 from e1. (See Existential Quantification, page ??)

UNIVERSE

The UNIVERSE function takes two(2) arguments, the first argument can be an equation or a variable (e1), the second argument must be a variable (v1). This function searches for and finds everyplace in e1 where v1 occurs. Every time v1 is found it is replaced by the logical AND of it's true branch and it's false branch. This function effectively removes v1 from e1.

PRINT_TREE

The PRINT_TREE function takes one(1) argument. The argument is printed to standard out in a nice, easily readable tree form. The argument is printed from the top down. The branch to the left is the true branch, the branch to the right is the false branch.

```
Example: PRINT_TREE( OR3(4, 5, -6) )
Prints to standard output:
------------------------------------------------------
---
                                6
               5                                T
        T             4
                  T       F
------------------------------------------------------
---
```

PPRINT_TREE

The PPRINT_TREE function takes one(1) argument. The argument is printed to standard out in a different form than PRINT_TREE. If the true and false branches are T and F then they are printed on the same line, otherwise, the true and false branches are indented from the top variable. The true branch is printed first and is followed by the false branch.

```
Example: PPRINT_TREE( OR3(4, 5, -6) )
Prints to standard output:
```

```
-----------------------------------------------------
ite 6
    ite 5
        T
        ite 4 T F
    T
-----------------------------------------------------
```

MINMAX

The MINMAX function takes at least 4 arguments. The first argument is the number of variables that will be in the resulting BDD. The second argument is the minimum number of variables in the BDD that need to be set to true to make the BDD evaluate to true. The third argument is the maximum number of variables in the BDD that can be set to true for the BDD to evaluate to true. The fourth argument is the list of variables that make up the BDD.

```
Example:PRINT_TREE( MINMAX(4, 1, 3, 13, 12, 11, 10)
Prints to standard output:
-----------------------------------------------------
---------
                                13
               12                                12
        11              T               T             11
    10       T                                      T      10
  F    T                                                T    F
-----------------------------------------------------
---------
```

ADD_STATE

The ADD_STATE function takes two(2) arguments. The first argument (e1) must be an equation and the second argument must be a positive integer (i1). ADD_STATE creates a new BDD with structure identical e1 but with each variable incremented by i1.

```
Example: ADD_STATE( AND(3, 4), 5 )
This will create the BDD - AND(8, 9)
```

7.1.3 Variable Names

7.1.4 Reusing Functions

7.1.5 Defines

7.1.6 Short Cuts

7.1.7 Example Input

```
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
```

7.2 CNF format

7.3 TRACE format

7.4 SMURF format

8 Preprocessing

III. The BDD Tool

Data is presented the ITE program via an input file. In this file BDD variables and BDDs are expressed using all Boolean functions, e.g., NOT, AND, NAND, OR, NOR, EQU, XOR, IMPLIES, etc. (See page ?? for a full list of commands and a description of each). Every BDD expressed may be reused when creating other BDD expressions. Input from this file continues until the specified number of user inputs has been processed (See page ??).

The ITE program processes the input given to the program, but the main tool exercised is the BDD Tool. It completes all of the manipulations and calculations necessary to build the BDD structure. The BDD Tool comes standard with many powerful BDD functions (See page ??). These functions give the user total control over their problem data.

8.1 Existential Quantification

8.2 Restrict

8.3 Strengthening

8.4 Dependent clustering

8.5 Cofactoring

9 Solver

The execution is handed to the solver after preprocessing is done. There are four different options controlled by the command line (ini file):

- SMURF brancher (default or when -b specified on the command line)
- BDD WalkSat (-w)
- WVF brancher (original pre-sbsat brancher)
- or it will start conversion of the internal problem representation into one of the specified formats.

9.1 SMURF Brancher

SMURF Brancher is given the problem as a collection of BDDs. Some of the BDDs are marked as special functions (AND, OR, AND=, XOR, ...). It converts non-special function BDDs and BDDs where the special function is not recognized (IMP, ...) into SMURF state machines. Special functions are handled using non-SMURF data structures in order to save memory space.

After conversion into SMURFs the process of solving the problem starts. There are .. major parts:

- heuristic - after processing all inferences without contradiction the new literal is chosen by looking at the heuristic values. Please see Section 10
- inference queue - every new inference is first put into the inference queue but the value of the inference is set as soon as it is found. The consequences of the inferences are processed in FIFO style (updating the SMURFs, special functions and lemmas).
- smurf update
- special function update
- lemma update
- lemmas
- backtracking

9.2 BDDWalkSat

(Sean!) Woohoo!

9.3 WVF Brancher

10 Solver heuristics

10.1 Johnson heuristic

10.2 Chaff like lemma heuristic

11 Output formats

For format description please see Section 7.

12 Examples on how to use SBSat

13 Reporting problems

Report bugs to <mkouril@ececs.uc.edu> or <franco@gauss.ececs.uc.edu>.