Short Introduction to the BED Tool

Poul Frederick Williams

IT University in Copenhagen
Glentevej 67, DK-2400 Copenhagen NV, Denmark
E-mail: pfw@it-c.dk

October 30, 1998

Abstract

This paper describes briefly the program BED for manipulating Boolean expressions. Only the basic features are discussed.

1 Introduction

The program BED was developed 1996-1998 by Henrik Reif Andersen, Henrik Hulgaard, and Poul Frederick Williams as part of a research project in formal verification at the Department of Information Technology. The program is capable of representing and manipulating Boolean expressions. The underlying data-structure is called a Boolean Expression Diagram (BED), which is a generalization of a Binary Decision Diagram (BDD). The BED program gives the user a command line interface where the user can enter and manipulate Boolean expressions.

This paper describes the BED program as seen from the users' point of view. It assumes a basic knowledge of BDDs and Boolean expressions.

2 How to Start the BED Program

Type

bed

at the UNIX prompt to start the BED program. It will start in the default mode, which means it will allocate about 4 MB of memory for the BED data structure and about 0.5 MB of memory for internal caches. You can specify the memory sizes by one or more command line options. The command

bed -b 32 -c 4

will give you 32 MB of memory for the BED data structure and 4 MB of memory for caches.

If you write

bed -f filename.com

the program will execute the commands in the file *filename.com* as if they had been typed by the user on the keyboard. The only difference is that each line must be terminated with a semicolon. This corresponds to using the *source* command, which will be mentioned later on. Likewise, if you write

bed filename.bed

the program will read the file *filename.bed* as if the command *read filename.bed* had been entered by the user. The command *read* will be discussed later.

When you start the BED program, it will prompt you for a command. The following sections describe the commands. You can view previously entered commands by using the up and down cursor keys. The command *help* gives you an overview of all the possible commands. If you want a more detailed help on a certain command, give the command name as argument to *help*:

help upall

You can clear all Boolean expressions in memory using the command

clear

You exit the program by typing *exit* or pressing CTRL-D.

The command *set* allows you to set options for the BED program. Type

set

to view the current options. The option *reductions* determines how Boolean connectives are handled internally. Default is *on*, but by writing

set reductions off

you can turn them off. When the reductions are turned on, the BED program tries to simplify the Boolean expressions you enter. This is done to minimize the number of vertices in the graph. It might be a good idea to turn the reductions off if you want to examine your expressions with the commands *print* and *view*. This makes it easier to recognize your expressions.

3 How to Enter Boolean Expressions

To enter a Boolean Expression, you first need to declare which variables you will be using. You do this with the command *var* by writing

var x y z

This will give you three variables x, y, and z. The variables will be ordered in order you wrote them. This means that any ROBDD which you now build will be ordered $x < y < z$. Further *var* commands add the new variables to the end of the existing order. The command

vars

lists of all the variables you have declared.

Next, you can build Boolean expressions using the *let* command:

let t = x and (y or z)

The node t will now represent the Boolean expression $x \wedge (y \vee z)$. You can use the Boolean operators shown in table 1. The operators are left associative. A zero (0) means false, while a one (1) means true. When you enter Boolean expressions, you may also use *true* and *false* instead of 1 and 0.

Operator	Syntax	Abbreviation
bi-implication	biimp	<=>
exclusive or	xor	!=
right implication	imp	=>
not right implication	nimp	
left implication	limp	<=
not left implication	nlimp	
or	or	|
not or	nor	
and	and	&
not and	nand	
it-then-else	*low* < *x* > *high*	
negation	not	!, ~
existential quantification	exists *x* . *r*	
substitution	[*x*:=*r*]	
reachability	reach(*I*, *T*)	

Table 1: Boolean operators allowed in the BED program. The operators are ranked after their precedence with the operators with the weakest bindings at the top. The reachability command requires that each next-state variable follows directly after its current-state variables in the variable ordering. I is the initial state and T is the transition relation.

The command

lets

lists of all the Boolean expressions you have entered. You can remove the Boolean expression represented by t by writing

unlet t

4 How to Transform Boolean Expressions to ROBDDs

Use the command

upall t

to transform the Boolean expression represented by t to an ROBDD with the current variable ordering.

You may change the variable ordering by using the command

order [z x]

It will place the mentioned variables first in the ordering. After execution of the command above, the ordering will be $z < x < y$. Remember to use the command *upall* to apply the new ordering to an ROBDD.

The command $verify$ provides an alternative way to transform BEDs into ROBDDs with the current variable ordering:

verify t

5 How to Examine Boolean Expressions

The command

print t

prints the nodes in the representation of t. For large BEDs, this might take some time. Use the command

count t

to find the number of vertices in t.

The command

view t

gives a graphical representation of the Boolean expression represented by t. This command uses the UNIX programs *dot* and *ghostview*. For large examples, this might take some time.

The command

anysat t

returns a satisfying variable assignments for t, while

anynonsat t

returns a non-satisfying assignment for t. To count the number of satisfying assignments for t write

satcount t

These three commands only work if t has been turned into an ROBDD first.

6 File Commands

Use the *write* command to save Boolean expressions on disk:

write [t1 t2 t3] > filename.bed

The Boolean expressions $t1$, $t2$, and $t3$ will be saved in the file *filename.bed*. Use a star ($*$) instead of [$t1$ $t2$ $t3$] to write all Boolean expressions currently in memory to a file. Use the *read* command to retrieve the Boolean expressions again:

read filename.bed

This will clear any Boolean expressions already in memory.

You can execute a series of commands by writing a script-file and use the *source* command to execute each command in the script:

source filename.com

where *filename.com* is a text-file with a series of commands separated by semicolons.

7 List of Commands

The BED program understands the following commands:

upall	Transforms a BED into an ROBDD.
upone	Lifts one or more variables up to one or more roots.
verify	Uses upone repeatedly to transform a BED into an ROBDD.
anysat	Determines a satisfying assignment.
anynonsat	Determines a falsifying assignment.
satcount	Determines the number of satisfying assignments.
eval	Evaluates a Boolean function.
count	Returns the number of nodes visited on a depth-first traversal.
var	Adds variables.
let	Adds a root for a Boolean expression.
unlet	Undeclares a list of roots.
set	Sets an option.
dot	Outputs a BED in the DOT format.
view	Shows a graphical representation of a BED.
print	Prints a node and all the nodes reachable from it.
read	Reads a BED from file.
write	Writes a BED root to file.
source	Reads and executes a script of commands.
vars	Shows the declared variables in order.
lets	Prints the roots of a BED.
stat	Prints statistical information.
support	Prints the support for a BED.
order	Sets the variable ordering used in upall and verify.
clear	Clears any existing BED in memory.
exit	Exits the BED program.
cd	Changes working directory.
ls	Lists contents of the working directory.
pwd	Prints working directory.
!	Sends the rest of the command line to a sub-shell (sh).
echo	Echos text to the screen.

Type *help* < *command* > for more information on individual commands

8 Example

This section shows how to use the BED program for solving a simple problem. We consider the problem of computing the reachable state space of the state machine shown in figure 1. It has four states: $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. The initial state is $(0,0)$.

To encode the states, we need two Boolean variables. We choose the variables x and y, and the coding: x is true iff the first number in the state name is 1, and y is true iff the second number is 1. To calculate the reachable state space, we need a set of "shadow" variables. We call these variables the next-state variables: xp and yp.

We are now ready to encode the transitions of the state machine. The command

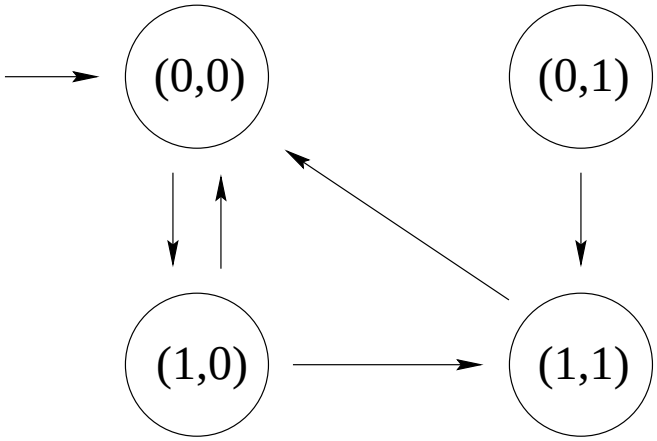

Figure 1: A state machine with four states. The initial state is $(0, 0)$.

var x xp y yp

gives us the necessary variables. Note that each next-state variable is placed right after the corresponding current-state variable. This is a requirement of the *reach* command, which we will use to calculate the reachable state space.

I is the initial state.

let I = !x and !y

T is the transition relation. It is a disjunction of all the transitions in the state machine.

let t1 = !x and !y and xp and !yp
let t2 = x and !y and !xp and !xp
let t3 = x and !y and xp and yp
let t4 = x and y and !xp and !yp
let t5 = !x and y and xp and yp

let T = t1 or t2 or t3 or t4 or t5

To calculate the reachable state space R, we use the command:

let R = reach(I, T)

The command

anynonsat R

gives us an assignment of the state variables for an unreachable state. In this case *anynonsat* returns $\neg x \wedge y$ corresponding to the unreachable state $(0, 1)$. The number of reachable states can be found by using

satcount R

and dividing the result by 2^n, where n is the number of state variables. In our case: $12/2^2 = 3$ reachable states.