

BDDC v2

A basic bdd-based logical calculator

Pascal RAYMOND

November 24, 2008, (rev. February 16, 2026)

BDDC is a tool for manipulating logical formula. It is based on a Binary Decision Diagram library, that means that each formula is internally transformed into a canonical form. In particular, if the formula (resp its negation) is a tautologie, BDDC will transform it into the "always true formula" (resp. the "always false formula").

All classical operators are provided. Moreover the tool provides a "functional-like" language that allows the user to define his (her) own logical operators.

1 Description

1.1 Basic features

The `bddc` utility implements an interactive processor for the evaluation of logical formula. It reads commands from the standard input, and prints the result to the standard output. The main commands are those which involve Boolean expressions: basically, `bddc` computes a bdd (binary decision diagram [1, 2]), which is a canonical representation of the expression.

For instance, if the user types the expression "`x + y.z ;`", the calculator interprets symbols `x`, `y` and `z` as Boolean arguments (i.e., basic propositions), and the whole expression as a Boolean function over those arguments. Note that "+" stands for logical *or*, and "." operator for logical *and*. The calculator builds the bdd corresponding to the expression, and then prints a polynomial representation of this canonical form; in the example, it simply outputs "`x + y.z`" (useful, isn't it?).

If the expression is a tautology, its bdd is the always-true function: for instance, if the user types "`x => (y => x);`" (`=>` is the logical implication), the result is "`true`".

1.2 Advanced features

The user may assign an expression to a variable, and then use this variable to build a new expression ("`>>`" is the prompt symbol of the processor):

```
>>f := (y => x);
x + -y
>>x => f;
true
```

Moreover, the user may define new operators on bdds. For instance :

```
>>union(A, B) := A + B;
--new function: union, arity: 2
>>union(x.y, t);
x.y + t
```

Indeed, this example is quite stupid. Defining new operators becomes interesting when using control operators and recursion (see 5.3, 6).

2 Lexical aspects

Lists are coma-separated (,).

Identifiers are any strings starting with a letter, and containing only letters, digits and under-score symbol, excepting the following keywords:

and	constrain	compare	cuts	cnf
del	dnf	else	exec	exist
false	forall	help	high	if
in	ite	implies	let	list
low	nor	not	or	print
quit	reed	restrict	root	set
size	syntax	then	true	xor

The sequence `//` starts a comment, which ends at the next new-line. The sequence `/*` starts a comment, which ends at the next `*/` sequence.

3 Commands

The processor is started by the command `bddc`, then the user enters an interactive session.

quit

exits the processor.

help

prints the list of commands.

exec "command-file"

reads commands from a file.

syntax

prints the syntax of logical expressions.

list

prints the list of arguments and variables in the current context (see 5.2).

exp;

builds the bdd corresponding to the expression in the current context, and outputs a polynomial equivalent to this bdd.

ident := exp ;

builds the bdd corresponding to the expression in the current context, assigns it to the variable *ident*, and outputs a polynomial equivalent to this bdd.

ident(identlist) := exp;

defines a function, which can be used later in logical expression.

print *ident*

print a polynomial equivalent to the bdd held by the variable *ident*. This command is obsolete: it is equivalent to type "*ident*";, since a identifier is a logical expression.

size *exp*;

prints the size of the bdd corresponding to *exp*. This size is expressed as a number of "bdd nodes" (see 5.1).

[dnf, cnf, reed] *exp*;

those commands are quite equivalent to typing "*exp*";, except that the output is printed (respectively) in disjunctive, conjunctive, or Reed-Muller normal form (see 5.5).

set [dnf, cnf, reed]

sets the output mode (disjunctive, conjunctive, or Reed-Muller normal form see 5.5).

4 Logical expressions

The calculator provides all the classical Boolean operators, plus some other specially dedicated to the bdd implementation. The syntax is infix, and the expression can be typed on more than one line, so don't forget the terminating symbol ";"!

4.1 Boolean operators

false

(or "0") always-false function.

true

(or "1") always-true function.

ident

reference to an argument or a variable (depending on the current context).

not *exp*

(or " $\neg exp$ ") complement.

exp* or *exp

(or " $exp + exp$ ") disjunction.

exp and exp

(or "*exp* . *exp*") conjunction.

exp xor exp

(or "*exp* <> *exp*") exclusive or.

exp => exp

implication.

exp = exp

equivalence.

ite(exp,exp,exp)

functionnal "if then else"; "*ite*(*f*,*g*,*h*)" is equivalent to "*f* . *g* + ~*f* . *h*".

forall identlist exp

universal quantifier.

exist identlist exp

existential quantifier.

(exp-list)

at most one true in the list.

nor exp-list

none is true in the list.

xor (exp-list)

exactly one true in the list. Warning: *xor* (*x*₁,*x*₂,...,*x*_{*i*}) is NOT "*x*₁ *xor* *x*₂ *xor* ... *xor* *x*_{*i*}" (except for *i*=2).

4.2 Control operators

Those operators allow the user to control the way expressions are evaluated.

compare(exp,exp)

returns 1 (the always-true function) if the two expressions are logically equivalent, 0 (the always-false function) otherwise.

implies(exp,exp)

returns 1 if the first expression implies the second one, 0 otherwise.

cuts(*exp*,*exp*)

returns 1 if the intersection of the two expressions is not empty (i.e. their logical and is not 0), 0 otherwise.

if *exp* then *exp* else *exp*

lazy "if": if the first operand is always-true (resp. always-false), the second (resp. third) operand is not evaluated; otherwise, it behaves as the `ite` operator.

let *ident* := *exp* in *exp*

local variable definition : the first expression is evaluated in a new context where *ident* is undefined. Then, the second expression is evaluated in a new context where the first result is associated to *ident*. The original context is then restored, and the second result is returned (see 5.2 for details).

4.3 Bdd decomposition

The following operators are specific to the bdd representation: in particular, they do depend on the internal order of the arguments.

Every formula "*f*" that actually depends on at least one argument (i.e. neither 0 nor 1) is equivalent to "*f* = if `root(f)` then `high(f)` else `low(f)`", where:

- `root(f)` is the least argument *f* actually depends on, according to the argument total order.
- neither `high(f)` nor `low(f)` depend on `root(f)`

See 5.4 for more details on Shannon decomposition and BDD.

root(*exp*)

the least argument which the expression depends on.

high(*exp*)

the "high" branch of the expression (i.e. the formula when the root-var is true).

low(*exp*)

the "low" branch of the expression (i.e. the formula when the root-var is false).

root(*exp-list*)

when called with more than one argument, the `root` function returns the minimal root-var of all the expressions. This generalisation is quite useful when defining recursive functions on the bdd structure.

supp(*exp*)

returns the "support" of the expression, that is, the "or" of all arguments the formula depends on.

4.4 Simplification modulo care-set

The following operators are specific to the bdd representation: in particular, they do depend on the internal order of the arguments.

It is often the case that a particular formula is manipulated under a set of assumptions, or "care-set".

restrict(exp,exp)

"restrict(f,g)" is a function equivalent to "f" if "g"; the result is simpler than "f" (less bdd nodes). The result is a function h satisfying:

$$if(f \wedge g = 0) \text{ then } h = 0$$

$$if(f \vee \neg g = 1) \text{ then } h = 1$$

Otherwise, h is some function satisfying:

$$(f \wedge g) \Rightarrow h \Rightarrow (f \vee \neg g)$$

N.B. The result is undefined if the second operand is 0.

constrain(exp,exp)

(also called the generalized cofactor) is a kind of "restrict" operation. The result is not garanted to be "smaller" than f , but it has an interesting (but not trivial) property: let $h = \text{constrain}(f,g)$

$$h(x) = f(x') \text{ where } x' = \text{"the closer point to } x \text{ in } g\text{"}$$

The precise definition of the "closer point ..." function is a little bit complicated (and depends on the variable ordering), but the main result is that the choice of x' **does not depend on** f .

N.B. The result is undefined if the second operand is 0.

4.5 Priorities

Expressions can be enclosed within parenthesis. Some priorities are used to resolve ambiguities:

- not, exist, forall,
- else
- and
- or
- => (right associative)
- =, xor (right associative)
- in

For instance:

```
let z := t = v in - exist x if y then x else z + z = y => z ;
```

is interpreted as:

```
let z := (t = v) in ((- (exist x (if y then x else z)) + z) = (y => z));
```

5 Appendix

5.1 Notes on BDD size

The tool uses a internal representation knowned as "signed binary decision diagrams" (SBDD). With this representation, a boolean function F and its negation $\neg F$ are (almost) represented by the same internal structure (they differ only on a boolean flag). With classic BDDs, the internal representation of F and $\neg F$ are disjoint. As a consequence, the cost of the negation is constant with SBDDs, since it is linear with classical BDDs. Moreover, it is known that this representation is always cheaper (in size and in time computation) than classical BDDs.

5.2 Notes on identifiers

The calculator deals with several kinds of identifiers:

arguments are the (global) free variables, i.e. identifiers that was once used without being first declared.

global variables are the identifiers introduced at the top level by a assignment command.

local variables are either formal parameters of a user-defined function, or identifiers introduced by a `let` operator.

The way an identifier is interpreted (i.e. the identifier's status) depends on the current context. Intuitively, at the top level, there is a global context, giving the status of arguments and global variables. The status of a global identifier is implicitly given by its first occurence: if it first appears in the left hand side of an assignment, the status is *variable*; if it first appears in an expression, the status is *argument*. One can type "list" to see the current global context :

```
>>x := a . b;
x := a . b
>>list
parameters are: a b
variables are: x
```

An argument cannot be re-used as a global variable; a variable, indeed, can be re-assigned:

```
>>a := x;
can't assign "a", not a variable
>>x := a + b;
x := a + b
```

Arguments are **globals**, while the scope of variables can be restricted using the `let ... in ...` operator.

```
>>y := let z := c + d in z;
y := c + d
>>list --note that z has disappeared, but not c and d:
arguments are: a b c d
variables are: x y
```

If `id` is a variable, `let id := ...` shadows the current value of `id`:

```
>>y;
c + d
>>z := let y := (let y := e.f in y.g) in x + y;
z := a + b + e.f.g
>>y;
c + d
```

Moreover, a local declaration shadows the status of the identifier:

```
>>a;
a
>>let a := e.f in a;
e.f
>>a;
a
```

5.3 Notes on user-defined functions

For the time being, no static verifications are made on user-defined functions. We can simply give the semantics for function calls: if the user has defined a function `foo(i1, ..., ik) := exp;`, then `foo(e1, ..., en)`, where `e1, ..., en` are correct expressions, is a correct expression equivalent to:

```
let i1 := e1 in
...
let in := en in
  exp;
```

In particular, a user-defined function may use free identifiers, but it is indeed quite dangerous: depending on the context call, those identifiers will be interpreted either as arguments, global variables, or even local variables of the current scope!

5.4 Notes on Shannon decomposition

The bdd decomposition is a special case of Shannon decomposition, where the choice of the decomposition variable is forced by the actual bdd representation. The general Shannon decomposition does not depend on the bdd structure, and can be obtained using logical operators; for all variable x and function f , we have $f = (x \wedge f1) \vee (\neg x \wedge f0)$, where (in bddc syntax):

```
f1 := exist x (x.f); -- or better: f1 := constrain(f,x);
f0 := exist x (-x.f); -- or better: f0 := constrain(f,-x);
```

5.5 Notes on output format

By default, the tool prints the results in a disjunctive normal form (sum of "ands"). The idea is to make the result as "readable" as possible. The quality of the "result" dramatically depends on the internal order of the arguments. For example, depending on the order of its arguments, the expression "if A then B else C" can produce different (but equivalent) polynomials, such as "A.B + -A.C + B.C" or "B.A + -A.C". Indeed, the second form is "better" since it is more concise, but the problem of printing the simplest polynomial equivalent to a given function is exponential. So, there is no guarantee on the "quality" of the print result.

The user can also prints the results in a conjunctive normal form (product of or's). This is the dual of the first form.

The Reed-Muller normal form consists in expressing the boolean functions as polynomials in "and" and "xor" (the always-true function "1" is also necessary). This form is canonical (modulo the commutativity and associativity of the operators). For instance:

```
>>reed (if A then B else C);
>>B.A <> A.C <> C
>>reed (not A);
>>A <> 1
```


This form is interesting (since it is canonical), but it can seem "strange" and unreadable for most people! Moreover, there is no "simple" relation between the size of the Reed-Muller form and a classical polynomial form: one form can be exponentially better or worse than the other:

```
>>A + B + C;
A + B + C

>>reed (A + B + C);
A.B.C <> A.B <> A.C <> A <> B.C <> B <> C

>>A <> B <> C;
A.B.C + A.-B.-C + -A.B.-C + -A.-B.C

>>reed (A <> B <> C);
A <> B <> C
```

6 Examples

6.1 Programming the negation

The following function implements the Boolean negation. Indeed, using this function is quite ineffective, since the built-in `not` operator does the same thing in constant time, but this simple algorithm shows the general structure of any unary bdd-based operator.

```
negation(a) :=
  if compare(a, true) then false
  else if compare(a, false) then true
  else ite( root(a), negation(high(a)), negation(low(a)));
```

In order to understand this algorithm, remain that any bdd `a` is either:

- `true`, i.e. the always true formula,
- `false`, i.e. the always false formula,
- a ternary "expression" of the form `ite(root(a), high(a), low(a))`, where `root(a)` is an argument¹, and `high(a)` and `low(a)` are two formulas not depending in `root(a)`.

The algorithm first treat the basic cases, by **comparing** the parameter to the constants `true` and `false`, and after the general case, where the parameter still depends on some argument. In this case, the function is recursively called on the corresponding high and low parts, and the results are combined under a test of the root argument.

6.2 Finding an implicant in a bdd

This is another example of unary function, that builds, if it exists, an monomial that implies `a` :

```
path(a) :=
  if compare(a, true) then true
  else if compare(a, false) then false
  else if compare(high(a), false) then (not root(a) and path(low(a)))
  else (root(a) and path(high(a)))
;
```

¹More precisely the identity formula associated to an argument

6.3 Programming the union

The following function implements the Boolean or. The only interest is in showing how to build a binary operator. We first treat the basic cases, where either *a*, *b* are constants. Then, we treat the case where both *a* and *b* are "binary" bdds. In this last case, we use the binary extension of the `root` operator to get the "minimal" argument between `root(a)` and `root(b)`, and call it `min_root`. Then, the recursive calls are made by decomposing both parameters according to `min_root`.

```
union(a, b) :=
  if (compare(a, true) or compare(b, false)) then a
  else if (compare(b, true) or compare(a, false)) then b
  else let min_root := root(a, b) in
    ite(min_root,
      union(constrain(a, min_root), constrain(b, min_root)),
      union(constrain(a, not min_root), constrain(b, not min_root))
    );
```

The use of the generalized cofactor in the recursive calls may seem a little bit tricky. It is justified by the fact that (for instance) either `constrain(a, min_root) = a` if `min_root ≠ root(a)`, or `constrain(a, min_root) = high(a)` if `min_root = root(a)`.

Annexe : syntax

This is the actual *non-ambiguous yacc* syntax, pretty-printed in ebnf with the tool `yacc2latex`. An abstract ambiguous syntax would be much simpler.

```
command ::= quit | exec <string> | help | syntax | echo | noecho | list
          | set ( reed | dnf | cnf )
          | ( reed | dnf | cnf ) exp ;
          | ( print | del ) ident
          | size exp ; | tree_size exp ; | unsigned_size exp ; |
          | TK_mcount ( quantifier_list ) exp
          | [ var_def | func_def | exp ; ]
ident_list ::= ident { , ident }
opexp_list ::= [ ident_list , ] opexp { , exp }
exp_list ::= ident_list
           | opexp_list
var_def ::= ident := exp ;
func_def ::= ident ( ident_list ) := exp ;
exp ::= ident
      | opexp
opexp ::= let ident := exp in exp
      | ( exp )
      | not exp
      | exp and exp
      | exp or exp
      | exp => exp
      | exp = exp
      | exp xor exp
      | if exp then exp else exp
      | forall ( quantifier_list ) exp
```

	forall <i>quantifier_list</i> <i>exp</i>
	exist (<i>quantifier_list</i>) <i>exp</i>
	exist <i>quantifier_list</i> <i>exp</i>
	ident (<i>opexp_list</i>)
	ident (<i>ident_list</i>)
	ident ()
	# (<i>exp_list</i>)
	nor (<i>exp_list</i>)
	xor (<i>exp_list</i>)
<i>quantifier_list</i>	::= { <i>ident</i> , } <i>ident</i>

References

- [1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6), 1978.
- [2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.