The IT University of Copenhagen

CLab 1.0 User Manual

Rune M. Jensen

IT University Technical Report Series

TR-2004-46

ISSN 1600–6100

August 2004

ISSN 1600–6100

ISBN 87-7949-068-9

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 København S
Denmark

Telephone:	**+45 72 18 50 00**
Telefax:	**+45 72 18 50 01**
Web	**`www.itu.dk`**

CLab 1.0 User Manual

Rune M. Jensen

Abstract

This document describes version 1.0 of CLab: a C++ library for fast backtrack-free and complete interactive product configuration using binary decision diagrams.

Contents

1 Getting Started

CLab is an open source C++/STL library for fast backtrack-free interactive product configuration. It encodes configurations in binary and uses reduced ordered Binary Decision Diagrams (BDDs) [1] to represent and reason about large configuration spaces. CLab utilizes the BuDDy BDD package [2] for handling BDDs. Instead of encapsulating this package, CLab works side by side with BuDDy as an advanced support tool. The BDDs generated by CLab can be printed, saved, and further manipulated using the numerous functions of the BuDDy package. This makes CLab suitable for research and education without compromising its ability to support real product configuration applications.

The implementation of CLab has been made as flat as possible to make it easy to alter the code and implement new functions. The library has two major functions: one that builds a BDD representing the configuration space of a declarative product model, and one that computes the set of possible ways a current partial configuration can be extended to a valid product. The latter function is fast (polynomial) and makes the interactive product configuration process complete and backtrack-free, since it allows the user to choose freely between any possible continuation of the partial configuration.

CLab 1.0 has been precompiled for Linux version 2.4. It may run under earlier and later Linux versions as well, and it should be fairly simple to port to other operating systems. To install, download the file `CLab10.tar.gz` from `www.itu.dk/people/rmj/clab/`. To unzip and untar the files execute the commands:

```
$ gunzip CLab10.tar.gz
$ tar xvf CLab10.tar
```

This creates the following directory structure:

```
CLab10
|-- src              : CLab C/C++ source files
|-- doc              : User manual
|-- examples         : CLab application examples
`-- buddy20          : Extended BuDDy 2.0 package
    |-- doc          : Buddy reference manual
    |-- examples     : Selfcontained BuDDy examples
    `-- src          : BuDDy C source files
```

To test the precompiled libraries for CLab and BuDDy go to `CLab10/examples/shirt` and compile the source files by running make:

```
$ make
```

This should produce the executable `shirt`. The effect of running `shirt` should be:

```
$ ./shirt
<var>: <valid assignments>
color: Black
print: MIB
size: Small
```

If you are unable to compile the example, or if the output is incorrect, you need to compile the source files of CLab and BuDDy manually. To compile CLab you need Flex and Yacc to be installed on your Linux system. Most systems have this by default. First, compile BuDDy. Go to `CLab10/buddy20/src` and do:

```
$ make clean
$ make
```

Second, compile CLab. Go to `CLab10/src` and do:

```
$ make clean
$ make
```

2 Quick Tour of CLab

This guided tour of CLab describes the shirt example in `CLab10/examples/shirt` that covers the main features of the library. The file containing the example code is `CLab10/examples/shirt/shirt.cc`:

```
 1 //////////////////////////////////////////////////////////////////////////
 2 // File  : shirt.cc
 3 // Desc. : Test file for CLab
 4 // Author: Rune M. Jensen
 5 // Date  : 7/19/04
 6 //////////////////////////////////////////////////////////////////////////
 7
 8 #include <string>
 9 #include <iostream>
10 #include <clab.hpp>
11 #include <bdd.h>
12 using namespace std;
13
14 int main()  {
15
16   CPR shirt("shirt.cp");
17
18   bdd solutionSpace = shirt.compileRules(cm_dynamic);
19
20   bdd  constraint = shirt.compile(Expr("size") == Expr("Small"));
21
22   solutionSpace &= constraint;
23
24   map< string, set<string> > va = shirt.validAssignments(solutionSpace);
25
26   cout << "<var>: <valid assignments>\n";
27     for ( map< string, set<string> >::iterator mit = va.begin(); mit != va.end(); ++mit)
28       {
29         cout << mit->first << ":";
30         for (set<string>::iterator sit = mit->second.begin(); sit != mit->second.end(); ++sit)
31           cout << " " << *sit;
32         cout << endl;
33       }
34   return 0;
35 }
```

This is an ordinary C++/STL source file that includes the header files of CLab (`clab.hpp`) and BuDDy (`bdd.h`). In line 16 a Configuration Problem Representation (CPR) object is constructed from the file `shirt.cp` containing a configuration problem description in the CP language (see Section 3). This file defines a simple T-shirt configuration problem:

```
1 ///////////////////////////////////////////////////////////
2 // File: shirt.cp
3 // Desc: CP file example.
4 //       Shirt configuration problem
5 // Auth: Rune M. Jensen
6 // Date: 7/19/04
7 ///////////////////////////////////////////////////////////
8
9 type
10   shirtColor {Black,White,Red,Blue};
11   shirtSize  {Small,Medium,Large};
12   shirtPrint {MIB,STW};
13
14 variable
15   shirtColor color;
16   shirtSize  size;
17   shirtPrint print;
18
19 rule
20   (print == MIB) >> (color == Black);
21   (print == STW) >> (size != Small);
```

A configuration problem consists of a set of variables with finite domains denoting the free parameters of the product and a set of rules defining the legal product configurations. For the T-shirt example there are three variables `color`, `size`, and `print` defining the color, size, and what to print on the T-shirt. The `color` variable is of type `shirtColor`. This is an enumeration type with elements {`Black,White,Red,Blue`}. Similarly the domains of `size` and `print` are {`Small,Medium,Large`} and {`MIB,STW`}. MIB and STW stand for "Men in Black" and "Save The Whales". The two rules in line 20 and 21 reflect the different requirements for these prints. Expressions on variables are ordinary conditional expressions of C, except that the pipe operator >> denotes logical implication (see Section 3 for details). The first rule says that the MIB-print must be on a black T-shirt (as one would expect for this movie commercial). The second rule says that the STW-print must be on a large or medium sized T-shirt (due to a large picture of a whale). In addition to enumeration types, it is possible to define range types. These are finite consecutive subsets of the integers. The only build-in type is Boolean.

The construction the CPR object in line 16 of `shirt.cc` involves parsing the `shirt.cp` file, type checking the rules and making a binary representation of each variable. In line 18 the two rules of the shirt problem are compiled into a BDD called `solutionSpace` that represents the set of legal configurations. This BDD is a Boolean function that given an assignment to the variables is true if and only if this assignment corresponds to a legal configuration.

In line 20 another BDD called `constraint` is constructed. This BDD represents all assignments that satisfy the expression `size == Small`. This constraint is assumed to have been chosen by a user in an interactive configuration session. To support further choices of the user, the solution space must be reduced to only contain assignments where `size` equals `Small`. This is done in line 22 by carrying out a BDD conjunction operation. The conjunction corresponds to making the intersection of the set of assignments represented by the `solutionSpace` and `constraint` BDD. See Section 4 for details on BDD-based configuration. Notice that this BDD operation is implemented by the BuDDy package and completely independent of CLab.

Finally the set of valid assignments for each variable in the restricted solution space is printed. The call to `validAssignments` in line 24 returns a map from variable names to sets of valid assignments for those variables. Surprisingly, there is just a single valid configuration left, since only an MIB T-shirt can be small according to rule 2, and due to rule 1, this T-shirt must be black:

```
$ ./shirt
<var>: <valid assignments>
color: Black
print: MIB
size: Small
```

If several configurations were possible in the T-shirt example, a restriction of the users choice to only valid assignments would ensure that the current configuration would be extended to some partial configuration of a total valid configuration. Hence interactive product configuration based on single variable assignments and the `validAssignments` function is backtrack-free. The user never has to redo a choice to ensure a valid configuration is reached. It is also complete, since the user in each iteration can pick any of the remaining valid configurations. The `shirt.cc` file is using the following make file:

```
 1 # ========================================================================
 2 # File: Makefile
 3 # Desc: Makefile for shirt
 4 # Auth: Rune M. Jensen
 5 # Date: 7/19/04
 6 # ========================================================================
 7
 8 CFLAGS =  -W -Wtraditional -Wmissing-prototypes -Wall
 9
10 LIBDIR1 = ../../buddy20/src
11 INCDIR1 = ../../buddy20/src
12
13 LIBDIR2 = ../../src
14 INCDIR2 = ../../src
15
16 OBJ =     shirt.o
17
18 CCFILES = shirt.cc
19
20 # ------------------------------------------------------------
21 # Code generation
22 # ------------------------------------------------------------
23
24 .SUFFIXES: .cc
25
26 .cc.o:
27   g++ -I$(INCDIR1) -I$(INCDIR2)  -g  -c  $<
28
29 # ------------------------------------------------------------
30 # The primary targets.
31 # ------------------------------------------------------------
32
33 shirt:  $(OBJ)
34  g++  -g $(CFLAGS) -o shirt $(OBJ)  -L$(LIBDIR1) -L$(LIBDIR2) -lclab -lfl -lm -lbdd
35  chmod u+x shirt
36
37 clean:
38  rm -f *.o core *~
39   rm -f shirt
```

There is nothing special about this make file except that it defines library and include directories for CLab and BuDDy. A peculiarity, however, is that the -lbdd argument must be placed last in Line 34 to avoid linker dependency errors to the BuDDy library.

3 CP Language Definition

The CP language has two basic types: *range* and *enumeration*. A range is a consecutive and finite sequence of integers. An enumeration is a finite set of strings. The Boolean type is the range from 0 to 1. Range and enumeration types can be defined by the user. A CP description consists of a *type declaration*, a *variable declaration*, and a *rule declaration*. The type declaration is optional if no range or enumeration types are defined:

cp	::=	[`type` {typedecl}] `variable` {vardecl} `rule` {ruledecl}
typedecl	::=	id `[` integer `..` integer `]` `;`
	|	id { idlst } `;`

vardecl ::= vartype idlst `;`

vartype ::= `bool`
| id

idlst ::= id {`,` idlst}

ruledecl ::= exp `;`

An identifier is a sequence of numbers, letters and the character “_” that does not begin with a number. An integer is a sequence of digits possibly preceded by a minus sign. The symbols `//` start a comment that extends until the end of the line. The syntax of expressions is given below:

exp ::= integer
| id
| − exp
| `!` exp
| (exp)
| exp op exp

op ::= `*` | `/` | `%` | `+` | `-` | `==` | `!=` | `<` | `>` | `<=` | `>=` | `&&` | `||` | `>>`

The semantics, associativity, and precedence of arithmetic, logical, and relational operators are defined as in C/C++. Hence, `!`, `/`, `%`, `==`, `!=`, `&&`, and `||` denote logical negation, division, modulus, equality, inequality, conjunction, and disjunction, respectively. The only exception is the pipe operator `>>` that denotes implication. The precedence and associativity is shown in Table 1. Notice that the convention of following C/C++ precedence causes the pipe operator to have higher precedence than is usual for logical implication. For this reason, the assignments in the two expressions in line 20 and 21 of `shirt.cp` are parenthesized.

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td><code>! -</code></td><td>right to left</td></tr>
<tr><td><code>* / %</code></td><td>left to right</td></tr>
<tr><td><code>+ -</code></td><td>left to right</td></tr>
<tr><td><code>>></code></td><td>left to right</td></tr>
<tr><td><code>< <= > >=</code></td><td>left to right</td></tr>
<tr><td><code>== !=</code></td><td>left to right</td></tr>
<tr><td><code>&&</code></td><td>left to right</td></tr>
<tr><td><code>||</code></td><td>left to right</td></tr>
</table>

Table 1: Precedence and associativity of operators.

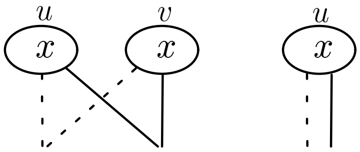
The semantics of an expression is the set of variable assignments that satisfy the expression. For example assume that the type of variable `x` and `y` is the range `[-4..2]`. The set of assignments to `x` and `y` that satisfies the expression `x + 2 == y` is then $\{\langle -4,-2\rangle, \langle -3,-1\rangle, \langle -2,0\rangle, \langle -1,1\rangle, \langle 0,2\rangle\}$. An assignment for which there exists an undefined operator in the expression is assumed *not* to satisfy the expression. Thus, the set of assignments to `x` and `y` that satisfies `x / y == 2` is $\{\langle -4,-2\rangle, \langle -2,-1\rangle, \langle 2,1\rangle\}$.

Conversion between Booleans and integers is also defined as in C/C++. True and false is converted to 1 and 0, and any non-zero arithmetic expression is converted to true. Due to these conversion rules, it is natural to represent the Boolean constants true and false with the integers 1 and 0.

4 BDD-Based Configuration

CLab uses BDDs to represent and reason about large configuration spaces. A BDD is a rooted directed acyclic graph representing a Boolean function on a set of linearly ordered Boolean variables. It has one or two terminal nodes labeled 1 or 0 and a set of variable nodes. Each variable node is associated with a Boolean variable and has two outgoing edges *low* and *high*. Given an assignment of the variables, the value of the Boolean function is determined by a path starting at the root node and recursively following the high edge, if the associated variable is true, and the low edge, if the associated variable is false. The function value is *true*, if the label of the reached terminal node is 1; otherwise it is *false*. The graph is ordered such that all paths respect the ordering of the variables.

A BDD is reduced such that no pair of distinct nodes u and v are associated with the same variable and low and high successors (Fig. 1a), and no variable node u has identical low and high successors (Fig. 1b). Due to these



(a) (b)

Figure 1: (a) nodes associated to the same variable with equal low and high successors will be converted to a single node. (b) nodes causing redundant tests on a variable are eliminated. High and low edges are drawn with solid and dashed lines, respectively

reductions, the number of nodes in a BDD for many functions encountered in practice is often much smaller than the number of truth assignments of the function. Another advantage is that the reductions make BDDs canonical [1]. Large space savings can be obtained by representing a collection of BDDs in a single multi-rooted graph where the sub-graphs of the BDDs are shared. Due to the canonicity, two BDDs are identical if and only if they have the same root. Consequently, when using this representation, equivalence checking between two BDDs can be done in constant time. In addition, BDDs are easy to manipulate. Any Boolean operation on two BDDs can be carried out in time proportional to the product of their size. The size of a BDD can depend critically on the variable ordering. To find an optimal ordering is a co-NP-complete problem in itself [1], but a good heuristic for choosing an ordering is to locate dependent variables close to each other in the ordering.

To use BDDs for configuration it is necessary to encode the set of valid configurations as a Boolean function. The arguments to this function is a Boolean representation of the configuration variables. The value of the function should be true only if the argument variables are assigned values that yield a legal configuration. It is simple to define a Boolean encoding of the configuration variables. Assume that variable domains contain successive integers starting from 0. For example we encode the enumeration $\{small, medium, large\}$ as $[0..2]$ and the range $[-4..2]$ as $[0..6]$. Let $l_i = \lceil \lg |D_i| \rceil$ denote the number of bits required to encode a value in domain D_i of variable i. Every value $v \in D_i$ can be represented in binary as a vector of Boolean values $\vec{v} = (v_{l_i-1}, \cdots, v_1, v_0) \in \mathbb{B}^{l_i}$. Analogously, every variable x_i can be encoded by a vector of Boolean variables $\vec{b} = (b_{l_i-1}, \cdots, b_1, b_0)$. Now, an assignment expression $x_i = v$ can be represented as a Boolean function given by the expression $b_{l_i-1} = v_{l_i-1} \wedge \cdots \wedge b_1 = v_1 \wedge b_0 = v_0$. For the T-shirt example we have, $D_2 = \{small, medium, large\}$ and $l_2 = \lceil \lg 3 \rceil = 2$, so we can encode $small \in D_2$ as 00 $(b_1 = 0, b_0 = 0))$, medium as 01 and large as 10.

The translation to a Boolean domain is not surjective. There may exist assignments to the Boolean variables yielding invalid values. For example, the combination 11 does not encode a valid value in D_2. Therefore we introduce a *domain constraint* that forbids these unwanted combinations $F_D = \bigwedge_{i=1}^{n}(\bigvee_{v \in D_i} x_i = v)$. Using the Boolean encoding of variables, rule i can be translated to a Boolean function r_i that is true for every assignment satisfying the rule. Recall that this also involves discarding assignments for which some operation is undefined. A Boolean function

S representing all valid assignments is given by

$$S = F_D \wedge \bigwedge_{i=1}^{n} r_i$$

When CLab compiles the rules of a CP description, it first builds a BDD for the domain constraint and for each rule and then conjoins these BDDs together to get a BDD representing S.

The BDD of the solution space of the T-shirt example described in Section 2 is shown in Figure 2. The variables `color`, `size`, and `print` are encoded using the Boolean vectors (x_1^1, x_1^0), (x_2^1, x_2^0), and (x_3^0). Each path in the BDD may encode one or more assignments. The leftmost path encodes two assignments $\langle x_1^1 = 0, x_1^0 = 0, x_2^1 = 0, x_2^0 = 1, x_3^0 = ?\rangle$. That is `color` = 00 = Black, `size` = 01 = Medium, and `print` = 0/1 = MIB/STW. Both of these assignments satisfy the rules of this problem and are thus valid configurations.

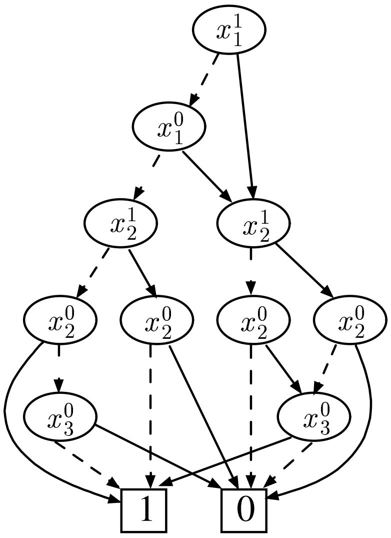


Figure 2: BDD of the solution space of the T-shirt example. Variable x_i^j denotes bit b_j of the Boolean encoding of product variable x_i.

5 Library Reference

CLab implements an expression class `Expr` and a Configuration Problem Representation class `CPR`.

5.1 Class Expr

The `Expr` class is a concrete type for building CP expressions within C++. Since CP expressions strictly follow the semantics, precedence, and associativity of C/C++ conditional expressions, the expressions of the `Expr` class are identical to CP expressions written within a CP description file. One must, however, take care to deal correctly with the automatic type conversion of C++. For instance, the following line of C++ code compiles successfully:

```
Expr e = "size" == "Small";
```

The result, however, will not be as expected. The compiler will resolve the overloading of the == operator by comparing the reference to two constant char pointers and return 0. Hence, `e` is an integer expression of 0 and not an equality test on variable `size` and enumeration element `Small`. To solve this problem leafs in the expression tree should be type casted:

```
Expr e = Expr("size") == Expr("Small");
```

In some cases, it may be fine to ignore incorrect casting:

```
Expr e = 2 + 3 + Expr("x");
```

The above example also compiles without problems, but `e` holds the expression $5 + x$ and not $2 + 3 + x$. The reason is that the left associativity of plus makes the type casting proceed from left to right. The first plus is therefore resolved to plus on integers and returns 5. The next plus, however, is due to the type casting of x resolved to plus on expressions on the `Expr` class. This causes an automatic cast of the left 5 to an integer expression. Expressions of class `Expr` are independent of any CP description. Expressions can therefore be written before any CPR object is defined. Expressions are only type checked when compiled by a CPR object.

Public Members:

```
Expr::Expr();
```

Default constructor.

```
Expr::Expr(const Expr& e);
```

Copy constructor.

```
Expr::Expr(int v);
```

Converts an integer to an integer expression.

```
Expr::Expr(std::string s);
```

Converts a string to an id expression.

```
Expr::Expr(char* s);
```

Converts a char string to an id expression.

```
Expr& Expr::operator=(const Expr& e);
```

Copy assignment operator.

```
Expr::~Expr();
```

Destructor.

```
std::string Expr::write()
```

Returns a string representation of the expression.

Nonmember Functions

```
Expr operator-(const Expr& e);
```

Returns the arithmetic negation of `e`.

```
Expr operator!(const Expr& e);
```

Returns the logical negation of `e`.

```
Expr operator>>(const Expr& l, const Expr& r);
```

Returns `l` implies `r`.

```
Expr operator||(const Expr& l, const Expr& r);
```

Returns `l` disjoined `r`.

```
Expr operator&&(const Expr& l, const Expr& r);
```

Returns `l` conjoined `r`.

```
Expr operator<=(const Expr& l, const Expr& r);
```

Returns `l` less than or equal to `r`.

```
Expr operator>=(const Expr& l, const Expr& r);
```

Returns `l` greater than or equal to `r`.

```
Expr operator<(const Expr& l, const Expr& r);
```

Returns `l` less than `r`.

```
Expr operator>(const Expr& l, const Expr& r);
```

Returns `l` greater than `r`.

```
Expr operator!=(const Expr& l, const Expr& r);
```

Returns `l` not equal `r`.

```
Expr operator==(const Expr& l, const Expr& r);
```

Returns `l` equal `r`.

```
Expr operator-(const Expr& l, const Expr& r);
```

Returns `l` minus `r`.

```
Expr operator+(const Expr& l, const Expr& r);
```

Returns `l` plus `r`.

```
Expr operator%(const Expr& l, const Expr& r);
```

Returns `l` modulus `r`.

```
Expr operator/(const Expr& l, const Expr& r);
```

Returns `l` divided by `r`.

```
Expr operator*(const Expr& l, const Expr& r);
```

Returns `l` times `r`.

5.2 Class CPR

The Configuration Problem Representation class (`CPR`) is the main class of CLab. A `CPR` object is constructed from a file containing a CP description. It is then possible to compile a BDD representing the set of valid configurations (the solution space) for the configuration problem and compile BDDs representing sets of variable assignments satisfying expressions of the `Expr` class. An important operation is to compute the set of valid assignments of each variable of a BDD representing a possibly reduced solution set of the `CPR` object. This operation returns a map from variable names to sets of assignments of that variable for which there exists some solution with the variable assigned to that value. For debugging purposes, it is possible to write a file with a readable content of a BDD representing a set of variable assignments of a `CPR` object. In addition, the internal state of a `CPR` object can be written to a string. See Section 6 for implementation details. In case of errors or warnings, an error function is called that in most cases causes the program to terminate. This behavior can be changed by providing a user defined error function.

The CLab library is used side by side with the BuDDy BDD package. The `CPR` class, however, needs BuDDy to be initialized. BuDDy will be initialized if necessary during the construction of a `CPR` object. If BuDDy is already running, the number of BDD variables may be increased to the number of Boolean variables needed to encode the configuration problem represented by the `CPR` object. `CPR` objects of entirely different configuration problems can co-exist.

It is possible to separate the often time consuming rule compilation and interactive product configuration into two independent programs. This is done by compiling the BDD of some CP file and then saving it using BuDDy's file-writing facility. A different program -implementing the interactive product configurator- first constructs a `CPR` object of the CP file, but does not compile the rules. Instead it loads the previously stored BDD and bases the interactive product configuration on this BDD.

Public Members:

```
CPR::CPR(std::string cpFileName);
```

This is the only constructor for `CPR` objects. It carries out the following operations:

- Parsing and type checking the rules of the CP file given in `cpFileName`,
- Initialization and extension of the number of variables of the BuDDy package if needed,
- Initialization of various internal data structures.

```
bdd CPR::compileRules(CompileMethod method = cm_dynamic);
```

Returns a BDD representing the set of valid configurations of the configuration problem. The argument `method` defines the compilation approach. There are three options:

Method	Description
`cm_static`	Conjoin the BDDs of the rules in the order they appear in the CP file.
`cm_dynamic`	Add the BDDs of the rules to a work list. In each iteration, conjoin the two smallest BDDs and add the result to the work list. Return the resulting single BDD in the work list.
`cm_ascending`	Sort the BDDs of the rules ascendingly according to their size. Conjoin the sorted BDDs from left to right.

The default method is `cm_dynamic`.

```
bdd CPR::compile(Expr expr);
```

Returns a BDD representing the variable assignments satisfying the expression `expr`. The expression is type checked before compilation.

```
std::map<std::string,std::set<std::string>> CPR::validAssignments(bdd sol);
```

Returns a map from variable names to sets of variable values represented by strings. A value is included in the set if there exists some configuration in the set of configurations represented by the BDD `sol` where the associated variable is assigned to that value. True and false are represented by the strings "1" and "0". The integers are represented by strings {...,"-1","0","1",...}. An enumeration element is represented by a string of its name.

```
~CPR::CPR();
```

Destructor of `CPR` objects. Deallocates internal data structures.

```
std::string CPR::writeBDDencoding();
```

Writes the BDD variable layout of the `CPR` object to a string.

```
void CPR::dump(std::string dumpFilename, bdd b);
```

Writes a readable dump to the file `dumpFilename` of the set of assignments stored in the BDD `b`. Each line of the dump corresponds to one or more assignments. If a line represents several assignments, one or more variables will have their values described in binary. In these binary encodings a "*" denotes a don't care (that is either true or false). It is necessary to know the value encoding of the variables to interpret these patterns. The value encodings are written by `writeBDDencoding` described above.

```
void setErrorFunc( void (*errorFunc) (int,std::string) );
```

Overwrites the default error function with the function pointed to by `errorFunc`. The `int` argument is the error type while the `std::string` argument is the error message. There are five types of errors:

Error type	Description
0	Warning
1	Parse error
2	Type checking error
3	Operating system error
4	CLab internal error

```
std::string CPR::write();
```

Writes a string representation of the internal state of the `CPR` object. See Section 6 for details.

6 Implementation

Clab is implemented in C/C++/STL and uses Flex and Yacc to compile CP descriptions. A flat software architecture has been chosen to reduce the time needed to understand the code and make changes. To further support development, the source code is well commented, and for each data structure there is a function for generating a string representing the information it holds. In this section, an overall description of the file structure and internal representation of `CPR` objects is given. This should be enough to get started working on the code. We refer the reader to the BuDDy documentation shipped with CLab for a description of the BuDDy source code.

6.1 Internal Representation of **`CPR`** Objects

A `CPR` object is represented by six data elements:

```
class CPR {

public:
  //...

private:
  CP* cpP;
  Symbols* symbolsP;
  Layout* layoutP;
  Space* spaceP;
  ValidAsnData* vadP;
  void (*error) (int,std::string);
};
```

The `cpP` data structure is an internal representation of the parse tree of the CP description. The `symbolsP` data structure is a collection of symbols of the configuration problem including type names, enumeration elements, and variable names. This information is used to type check the type, variable, and rule declarations. The `layoutP` data structure contains the binary encoding of types and the translation of variables to vectors of BDD variables (the BDD layout). The `spaceP` data structure contains a BDD representing the domain constraints described in Section 4. The `vadP` data structure is used by the specialized BDD operation implemented in the BuDDy package to compute valid assignments efficiently. Finally `error` is the error function used by the `CPR` object.

6.2 File Structure

The CLab source files are in `CLab10/src`. The file structure is classical modular. Each C/C++ file implements one a more related classes and functions declared in the associated header file. The library is a mixture of C and C++ files due to the use of Flex and Yacc. It also contains Flex and Yacc definition files for the CP language. A description of each file in `CLab10/src` is given below.

File(s)	Description
`Makefile`	Make file for compiling the CLab library.
`depend.inf`	Header file dependency file included by `Makefile`.
`cp.l`	CP language token definition file for Flex.
`lex.yy.c`	Lexer produced by Flex.
`cp.y`	CP Language syntax file for Yacc.
`y.tab.c/h`	Parser produced by Yacc.
`common.cc/hpp`	Constant definitions and functions used through out CLab.
`set.cc/hpp`	Template functions for set manipulation.
`cp.cc/hpp`	Data structure of the internal representation build by Yacc.
`symbols.cc/hpp`	Data structure of type, enumeration, and variable symbols of a CP description. Member functions for type checking rule declarations and expressions of the `Expr` class.
`layout.cc/hpp`	Data structure of the BDD layout of the CP description. Contains the binary representation of types and the mapping from configuration variables to BDD variable vectors encoding the values of the variables. Member functions for printing the encoding.
`space.cc/hpp`	Data structure containing a BDD of the domain constraint. Member functions for compiling rules and expressions of the `Expr` class.
`dump.cc/hpp`	Functions for writing the assignments of a BDD to a file.
`clab.cc/hpp`	Header file for the CLab library. The default error function is implemented in `clab.cc`.

Acknowledgments

Thanks to the VeCoS group at the IT University of Copenhagen for valuable discussions on this work. The T-shirt example is due to Erik Van Der Meer.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8:677–691, 1986.

[2] J. Lind-Nielsen. BuDDy - A Binary Decision Diagram Package. Technical Report IT-TR: 1999-028, Institute of Information Technology, Technical University of Denmark, 1999. `http://cs.it.dtu.dk/buddy`.