

BOOLVAR/PB v1.0, a java library for translating pseudo-Boolean constraints into CNF formulae

Olivier Bailleux, Université de Bourgogne

March 22, 2011

Abstract

BOOLVAR/PB v1.0 is an open source java library dedicated to the translation of pseudo-Boolean constraints into CNF formulae. Input constraints can be categorized with tags. Several encoding schemes are implemented in a way that each input constraint can be translated using one or several encoders, according to the related tags. The library can be easily extended by adding new encoders and / or new output formats. It is available at <http://boolvar.sourceforge.net/>.

1 Introduction

Let us begin by an introductory exemple, which consists in translating the constraint $5x_1 + 3\overline{x_2} + x_3 \leq 8$ into a CNF formula. At first, an input model must be created as follows:

```
InputModel m = new InputModel();
```

Then, the three variables x_1, x_2, x_3 can be created as an array of instances of the class `Variable`.

```
Variable[] x = new Variable[3];
for(int i=0; i<3; i++) x[i] = new Variable();
```

The input constraint is composed of literals that can be produced from the variables. Each literal is an instance of the class `Literal`.

```
Literal[] lits = new Literal[3];
lits[0] = x[0].getPosLit();
lits[1] = x[1].getNegLit();
lits[2] = x[2].getPosLit();
```

Before to create the input constraint, we have to define the coefficients related to each literal and the tag that will be assigned to this constraint. Note that it is allowed to assign several tags to the same constraint. For example, `setTags(1,3)` will assigns the tags 1 and 3 to the upcoming constraints that will be created until the next call of `setTags`.

```
int[] coeffs = {5,3,1};
setTags(1);
```

To complete the building of the input model, it only remains to create the input constraint, which is simplified by using the static factory method `makeLeq`, and add it to the model.

```
Constraint q = makeLeq(coeffs,lits,8);
m.addConstraint(q);
```

At this time, the input model is created. It can be print to the screen for verification purpose.

```
System.out.println(m.toString());
```

Now, the input model must be translated to an output problem, namely a CNF formula. This suppose to create an instance of the class `CNFProblem` in the following way:

```
OutputProblem out = new CNFProblem();
```

Now, suppose that we want to encode the input constraint (which is marked with the tag 1) redundantly with two encoders. We have to create an instance of each of these encoders, and to assign these encoders to the tag 1. As a consequence, each constraint of the input model tagged with 1 will be translated using these two encoders.

```
Encoder2cnf direct = new CNFdirectEncoder();
Encoder2cnf bdd = new CNFbddEncoder();
out.assignEncoder(1,direct);
out.assignEncoder(1,bdd);
```

The translation process will be achieved by reading the input model thanks to the method `read` of the output problem.

```
out.read(m);
```

The resulting CNF formula is given in DIMACS format by the method `getOutput`.

```
System.out.print(out.getOutput());
```

For example, if only the encoder `CNFdirectEncoder` is used, the result is the following:

```
p cnf 3 1
-1 2 -3 0
```

This encoder produces an exponential number of clauses in the general case, but can produce quite compact outputs from small input constraints. The currently available encoders will be described in section 3.

To conclude this brief presentation, let us mention that `BOOLVAR` also provide a class `PBproblem` and an encoder `PBbasicEncoder` allowing to produce the output problem as an instance of pseudo-Boolean satisfiability with the OPB format. This allows a same input problem to be solved either using a SAT solver and a pseudo-Boolean satisfiability solver, in order to compare the performances and the relevance of the two approaches.

2 Description

This section presents the main aspects and resources of `BOOLVAR/PB` from the user side and as well as a short description of its internal structure.

2.1 Architecture

Figure 1 presents the architecture of the `BOOLVAR/PB` library, which can be decomposed in three parts: the input block, the output block, and the internal block.

2.1.1 The input block

This part includes the classes allowing the user to create and specify an input problem.

InputModel

The container for the input constraints. Its constructor must be used to create a new input problem where the input constraints will be added thanks to the method `addConstraint`.

Constraint

An interface requiring that any input constraint implements a method `addTag`, which allows to assign tags to constraints.

GenericConstraint

An abstract class which implements the tag management system, which is the same for any input constraint.

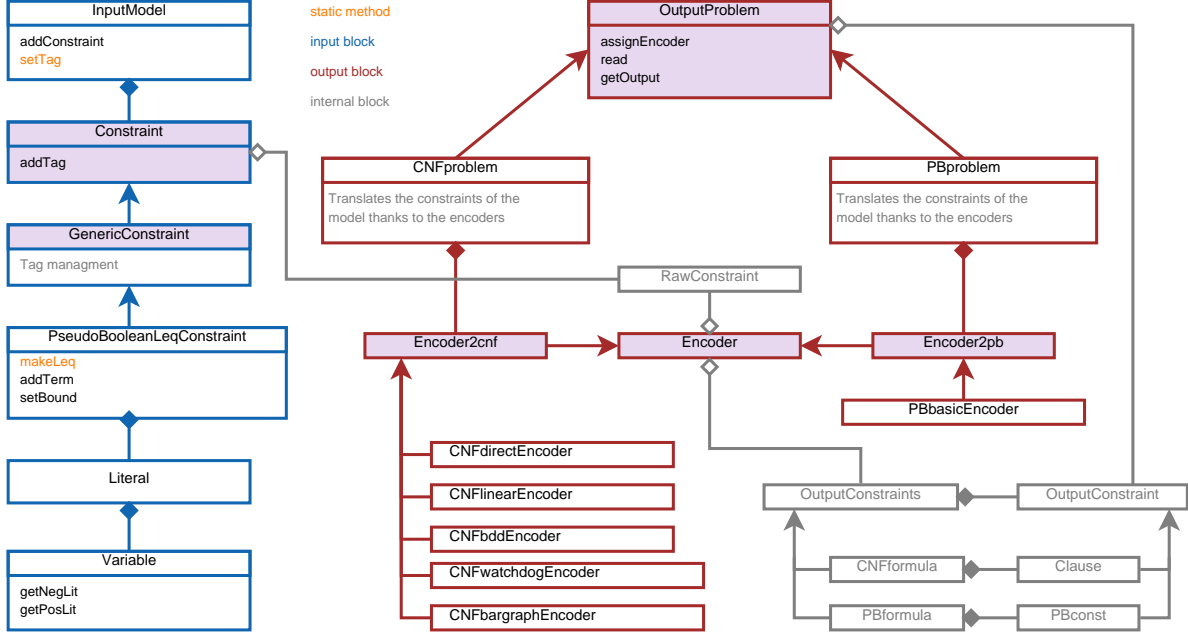


Figure 1: The architecture of BOOLVAR/PB

PseudoBooleanLeqConstraint

A pseudo-Boolean inequality constraint of the form $a_1x_1 + \dots + a_nx_n \leq b$, where x_1, \dots, x_n are propositional literals (i.e., Boolean variables or negated Boolean variables), a_1, \dots, a_n and b_n are positive integers. The instances of this class can be created in two ways : (1) using the provided constructors and building methods, or (2) using the static factory method `Boolvar.makeLeq`.

Literal

The building block for input constraints. A literal can be produces either by using the constructor of this class, or by using the methods `getNegLit` and `getPosLit` of the class `Variable`.

Variable

A representation for the propositional variables that are used both in the input and in the output constraints.

2.1.2 The output block

This part includes the classes allowing the user to specify the output problem, as wall as the way to produce this problem from the input constraints.

OutputProblem

This interface specify the methods that must be implemented in any kind of output problems: `assignEncoder`, `read`, and `getOutput`.

CNFproblem

A representation for CNF output problems. Basically, any instance of this class contains a set of encoders and a string that will be receive the result of the translation. The internal methods ensures the translation process.

PBproblem

A representation of a pseudo-Boolean output problem in the same way of the class **CNFproblem**. This class allows to produce an instance of the input problem in a form that allows to solve it using a pseudo-Boolean satisfiability solver.

Encoder

This interface represents an encoder, i.e., a class that contains the resources for translating constraints.

Encoder2cnf

This interface represents an encoder which produces a CNF formula as output. The available implementations will be presented in section 3.

Encoder2pb

This interface represents an encoder which produces a list of pseudo-Boolean constraints as output.

PBbasicEncoder

This encoder puts the input problem in a format that allows it to be solved using a pseudo-Boolean satisfiability solver.

2.1.3 The internal block

This part includes the internal resources allowing the communication between the input and the output block, as well as the implementation of the translation process.

RawConstraint

The internal representation for input constraints. Any class which implements the interface **Constraint** must provide a method producing an array of such internal constraints. Currently, any instance of the class **PseudoBooleanLeqConstraint** produces only one instance of **RawConstraint**, but a class **PseudoBooleanEqConstraint** could be implemented in a way to manage input constraints of the form $a_1x_1 + \dots + a_nx_n = b$. Such a constraint could either be represented as two internal inequality constraint or one internal equality constraint.

OutputConstraint

A generic output constraint.

Clause

Implementation of **OutputConstraint** as a propositional clause. Contains resources that can be used by the CNF encoders to produce output clauses, which will be converted into strings by the class **CNFproblem**.

PBconst

Implementation of **OutputConstraint** as a pseudo-Boolean constraint. Contains resources that can be used by the PB encoder to produce output pseudo-Boolean constraints, which will be converted into strings by the class **CNFproblem**.

OutputConstraints

A set of output constraints resulting from the translation process, which are aimed to be converted into strings to produce the output problem.

CNFformula

A set of Clauses represented as instances of the class **Clause**.

PBformula

A set of pseudo-Boolean constraints represented as instances of the class **PBconst**.

2.2 User resources

This section presents the main classes and methods allowing one to implement the resources provided by `BoolVar/PB`.

2.2.1 Input block user resources

`InputModel.InputModel()`

Create a new input model, which can be seen as a container for the input constraints.

`Variable.Variable()`

Create a new propositional variable, which is aimed to be used in an input constraint.

`Variable.getPosLit()`

Returns a positive literal v from the current variable v . If the literal v does not exist, it is created thanks to the constructor of the class `Literal`, else the reference of the existing literal is returned. v is aimed to be used in input constraints.

`Variable.getNegLit()`

Returns a negative \bar{v} literal from the current variable v . If \bar{v} does not exist, it is created thanks to the constructor of the class `Literal`, else the reference of the existing literal is returned. \bar{v} is aimed to be used in input constraints.

`Variable.getLit(boolean sign)`

Returns a literal v or \bar{v} , according to the value of `sign`, from the current variable v . If the required literal does not exist, it is created thanks to the constructor of the class `Literal`, else the reference of the existing literal is returned.

`Literal.Literal(Variable v, Boolean s)`

Create a new literal, which is aimed to be used in an input constraint. In order to avoid redundancies, literals must be preferably created thanks to the methods `getNegLit` and `getPosLit` provided by the class `Variable`.

`BoolVar.setTag(int tag1)`

`BoolVar.setTag(int tag1, int tag2)`

`BoolVar.setTag(int tag1, int tag2, ..., tag4)`

Sets the tags that will be assigned to the input constraints that will be created before the next call of `setTag`. The tags are arbitrary integers that will be assigned to encoders in a way to specify which encoder(s) must be used for translating each input constraint.

`BoolVar.makeLeq(int[] c, Literal[] l, int b)`

`BoolVar.makeLeq(BigInteger[] c, Literal[] l, BigInteger b)`

Create a new pseudo-Boolean inequality constraint $c[0]l[0] + \dots + c[n-1]l[n-1] \leq b$ (where n is the size of the arrays `c` and `l`) from an array `c` of coefficients, an array `l` of literals, and a bound `b`. The resulting constraint is aimed to be added to the input model thanks to the method `InputModel.addConstraint`.

`InputModel.addConstraint(Constraint q)`

Adds a new constraint to the current input model.

2.2.2 Output block user resources

`PBproblem.PBproblem()`

Creates a new output problem as a pseudo-Boolean satisfiability problem.

`PBproblem.PBproblem()`

Creates a new output problem as a propositional satisfiability problem.

`PBproblem.assignEncoder(int tag, Encoder x)`

Assigns the encoder `x` to the given tag. Several encoders can be assigned to the same tag by multiple calls to this method.

`PBproblem.read(InputModel m)`

Reads the input model `m` in a way to translate each input constraint of `m` with the related encoders. This method must be used only one time, after all the input constraints are added to the input model.

`PBproblem.getOutput()`

Returns the output problem as a string. This method must be called after the method `InputModel.read`.

3 Available encodings

The current version of `BOOLVAR/PB` includes 5 CNF encoders for pseudo-Boolean inequality constraints. The underlying encoding methods can be classified in different categories with respect to the size of the output formula and the inference power of unit propagation (which is the basic filtering technique used in the SAT solvers) on this formula.

Any CNF encoding which produces a CNF formula polynomially sized (exponentially sized, respectively) with respect to the number of variables in the input constraint is said to be *polynomial* (*exponential*, respectively). Any CNF encoding is said to be a PAC (like propagating arc consistency) encoding if and only if applying unit propagation on the resulting formula fixes the same variables as restoring arc consistency on the corresponding input constraint. It is said to be a PIC (like propagating inconsistency) encoding if and only if applying unit propagation on the resulting formula produces the empty clause if restoring arc consistency on the corresponding input constraint detects an inconsistency. Any PAC encoding is necessarily a PIC one.

3.1 CNFdirectEncoder

This simple exponential and PAC encoding is briefly described in [2]. It can be seen as a variant of the BDD based encoding introduced in [1] where each path of the BDD is encoded with a clause in a way that no additional variable is required.

3.2 CNFbddEncoder

This exponential PAC BDD-based encoding was introduced in [1]. It generally produces a smaller formula than the direct encoding, because it uses additional variables corresponding to each node of the BDD, which allows to factorize identical subgraphs. Unlike the direct one, this encoding is polynomial for cardinality constraints, i.e., when all the coefficients are 1.

3.3 CNFlinearEncoder

This linear encoding is described in [4]. It is inspired from the encoding introduced in [5]. It is neither PAC nor PIC. Its main interest is the size of the produced formulae.

3.4 CNFwatchdogEncoder

This encoding, introduced in [2], is both polynomial and PAC (then PIC), but can sometimes produce output formulae of prohibitive size.

3.5 CNFbargraphEncoder

This is a variant of the watchdog encoding, also presented in [2], which is PIC but not PAC, and produces smallest formulae.

4 A commented example

As an exemple, we will encode the bin-packing problem, where n objects, each of them with a weight $w_j, 0 \leq j \leq n-1$, must be put into m boxes with capacities $c_i, 0 \leq i \leq m-1$, in such a way that each object occurs in exactly one box and the sum of the weights of all the objects belonging to any box does not exceed the capacity of this box.

Each instance of this problem will be represented thanks to a matrix v of Boolean variables, where $v_{i,j} = 1$ means that the object j is in the box i .

There are two kinds of constraints, namely:

- the unicity constraints, which ensures that each object belong to exactly one boxe:

$$\forall j \in 0..n-1, \sum_{i=0}^m v_{i,j} = 1$$

- the capacity constraints, ensuring that the sum of the weights of the objects in any box does not exceed the capacity of this box:

$$\forall i \in 0..m-1, \sum_{j=0}^n w_j v_{i,j} \leq c_i$$

The bin-packing problem can be encoded with BOOLVAR in the following way. The integer arrays **weights** and **capacities** are supposed to contain the weights of the objects and the capacities of the boxes, respectively.

1. Create the input model.

```
InputModel p = new InputModel ();
```

2. Create and initialize the matrix of domain variables.

```
Variable [][] v = new Variable[m][n];
for(int i=0; i<m; i++)
    for(int j=0; j<n; j++)
        v[i][j] = new Variable ();
```

3. Create the unicity constraints and assign them the tag 1 with the method **setTag**. Each equality constraint $v_{0,j} + \dots + v_{m,j} = 1$ is encoded as two inequality constraints $v_{0,j} + \dots + v_{m,j} \leq 1$ and $\overline{v_{0,j}} + \dots + \overline{v_{m,j}} \leq m-1$.

```
setTags(1);
int[] coeffs = new int[m];
for(int i=0; i<m; i++) coeffs[i]=1;
for(int j=0; j<n; j++)
{
    Literal[] poslits = new Literal[m];
    Literal[] neglits = new Literal[m];
```

```

    for(int i=0; i<m; i++)
    {
        poslits[i] = v[i][j].getPosLit();
        neglits[i] = v[i][j].getNegLit();
    }
    p.addConstraint(makeLeq(coeffs, poslits, 1));
    p.addConstraint(makeLeq(coeffs, neglits, m-1));
}

```

4. Create the capacity constraints and assign them the tag 2.

```

setTags(2);
for(int i=0; i<m; i++)
{
    Literal[] lits = new Literal[n];
    for(int j=0; j<n; j++)
        lits[j] = v[i][j].getPosLit();
    p.addConstraint(makeLeq(weights, lits, capacities[i]));
}

```

5. Create the output problem.

```

OutputProblem out = new CNFProblem();

```

6. Create the two encoders that will be used to translate the input constraints: the BDD encoder will be used to translate the unicity constraints (tagged with 1), and the bargraph encoder will be used to translate the capacity constraints (tagged with 2).

```

Encoder2cnf bdd = new CNFbddEncoder();
Encoder2cnf bg = new CNFbargraphEncoder();
out.assignEncoder(1, bdd);
out.assignEncoder(2, bg);

```

7. Read the input model, which runs the translation process, and print the result as a string.

```

out.read(p);
System.out.println(out.getOutput());

```

5 Perspectives

The following evolutions are planned.

Predicting the size of the input formulae

The size of the formula resulting from the translation of each input constraint is a critical parameter for the choice of the encodings. The interface **Encoder** will include a method providing this information.

Automatic choice of the encoders

Assigning a dedicated tag to any constraint will ensure that the encoder to use for translating this constraint will be automatically selected accordingly to both the size of the output formula and the inference power of unit propagation on this formula.

Clauses as input constraints

It will be allowed to add clauses as input constraints. These clauses will be directly added to the output problem, in a way to allow to deal with input problems that are specified with both clauses and pseudo-Boolean formulae.

Supporting new encodings

Some encodings dedicated to cardinality constraints will be added. Because cardinality constraints are a special case of pseudo-Boolean ones, the pseudo-Boolean encoders can of course deal with cardinality constraints. But there exists specific encodings which could be more efficient and/or compact for cardinality constraints.

In addition, some of the already implemented encodings could be improved and / or hybridized in a way to reduce the size / efficiency ratio of the resulting output formulae.

Coupling with a solver

The goal is to provide the resources for solving the output problem thanks to the solver sat4J [3], in a way to build stand alone applications.

References

- [1] Olivier Bailleux, Yacine Bouffkhad, and Olivier Roussel. A translation of pseudo boolean constraints to sat. *JSAT*, 2(1-4):191–200, 2006.
- [2] Olivier Bailleux, Yacine Bouffkhad, and Olivier Roussel. New encodings of pseudo-boolean constraints into cnf. In *SAT*, pages 181–194, 2009.
- [3] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7:59–64, 2010.
- [4] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into sat. *JSAT*, 2(1-4):1–26, 2006.
- [5] J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 1968.