On Finding Counter-Models: Approach Based on Instantiating Abstract Sets

Daniel Crowley, Daniel Le Berre, Olivier Roussel, Yakoub Salhi

On Finding Counter-Models: Approach Based on Instantiating Abstract Sets

BLaSST ANR-21-CE25-0010, Deliverable D4.1
SMT-Solver Integrating SAT-Based Techniques

Daniel Crowley, Daniel Le Berre, Olivier Roussel, Yakoub Salhi

Univ. Artois, CNRS, UMR 8188, Centre de Recherche en Informatique de Lens (CRIL), F-62300 Lens, France
{crowley, leberre, roussel, salhi}@cril.fr

1 Introduction

The tool *set-sizing-for-counter-models* was designed for aiding in finding counter-models to B language machines by establishing adequate sizes for sets that are abstract in the original specification. The canonical intention of the authors was to use *set-sizing-for-counter-models* for preprocessing before using SMT-solvers on its output.

The development of *set-sizing-for-counter-models* has been a part of the Enhancing B Language Reasoners Using SAT and SMT Techniques (BLaSST) project, funded by the ANR and involving Inria in Nancy, CRIL in Lens, the ClearSy company, and the Montefiore Institute in Liège. The software is available under the GNU Lesser General Public License.[1]

2 Set Sizing

The B language is founded on set theory and proves the validity of abstract B language machines or refinements of abstract machines or of other refinements. A machine can be populated with abstract or concrete sets, where the concrete sets' sizes are set by the length of the enumeration of their elements. Abstract sets' sizes are unknown and the verification of a machine requires for the proof to hold for every possible size of the abstract sets. This situation can require very large search spaces.

Some restrictions are placed on the sizes of the sets in the `PROPERTIES` section of a machine. Only if these `PROPERTIES` obtain can a machine be shown to be invalid. The purpose of *set-sizing-for-counter-models* is to seek precise

[1] https://github.com/crillab/set-sizing-for-counter-models

cardinalities for the abstract sets. **With the supposition that constraints regarding the sizes of sets are put forth in the `PROPERTIES` section and not in other sections for premisses,** *set-sizing-for-counter-models* claims to provide counter-examples to B machines by instantiating the abstract sets with concrete sets of sizes suitable for satisfying the `PROPERTIES`, and finding that the goals fail to hold. This amounts to encoding and satisfying the constraints in all of the Define tags in a POG file named "ctx", "lprp", "inprp", or "sets".

When a machine is invalid, it is good to know so and to get an instance of the machine failing. Instantiating the abstract sets in the premisses can make the task of finding such counter-models easier.

3 How *set-sizing-for-counter-models* Works

3.1 Pseudo-Boolean Encoding

This software serves as a preprocessing step using SAT-solving before passing the machine to an SMT-solver. This is one way for SAT-solving techniques to help SMT-solvers in finding counter-models to B language machines. The chosen way to do this for *set-sizing-for-counter-models* was to translate the inputs to OPB format, which is read by pseudo-Boolean solvers. More specifically, they are translated to the restricted form used for the pseudo-Boolean solver competition in 2024 [3]. This is one possible avenue for integrating SAT-solving techniques, and pseudo-Boolean reasoning [4] is one appropriate choice for putting cardinality to the fore.

3.1.1 Linear Pseudo-Boolean Problem

In general, a linear pseudo-Boolean problem instance is a conjunction of constraints of the following form:

$$\sum_i c_i x_i \bowtie \alpha \tag{1}$$

All of the xs are propositional variables that, in each pseudo-Boolean constraint, take either the value of 0 or 1 according to whether the variable is valuated as true or false. Each integer constant c_i is the multiplicative coefficient of each corresponding x_i, and α, the degree of the constraint, is an integral constant that is compared to the sum of the products of the variables and their coefficients. Each c_i and α can be positive or negative. Non-linear pseudo-Boolean constraints additionally admit the product of variables, but such constraints are unnecessary for this encoding.

The $\bowtie$ in Clause 1 can sometimes represent any comparison of equality or directed inequality, but in the restricted format used for pseudo-Boolean solver the possibilities are reduced to the operations of equality and being greater than or equal to ($=$ or $\geq$).

Another notable difference between the general and restricted formats is that negative literals of propositional variables cannot appear in the restricted

format. These restrictions require at times extra steps in the encoding, but the restricted encoding is favoured in order to simplify as much as possible the work of the pseudo-Boolean solver.

As stated, the format describes instances of the pseudo-Boolean satisfiability problem, and such files in the OPB format tend to be given the extension `.pbs`. The aim of such solver is to find some interpretation of the variables of the instance that makes all of the constraints true. The more difficult optimization problem additionally requires the solver to find an interpretation that minimizes a cost function. The format for optimization instances, usually given the extension `.pbo` when in OPB format, differs from a `.pbs` file by including as well a single line defining the cost function.

3.1.2 Encoding `SETS`

The sets of the B language machine are encoded as `.pbo` files elementwise. When the upper bound is k, the set is given an array of k variables to represent possible elements.

A set in the original specification could be either abstract without any information about the cardinality that it must ultimately have, or concrete where its elements are enumerated and its cardinality can be gathered thereby. When a set is abstract, a constraint that the unweighted sum of its variables be at least one is included because, in B language machines, abstract sets are non-empty by design. On the other hand, concrete sets are constrained to have exactly as many elements as are enumerated, so the unweighted sum of its variables is equal to that number of enumerated values.

The cost function is determined based on the `SETS` section alone, and is realized by fleshing out the following schema:

$$min : \sum_{a \in AbsSets} \sum_{i=1}^{k} x_{ai}. \tag{2}$$

Here, k is the upper bound on how large an abstract set is allowed to be, so the sum is over all of its allotted variables. The motivation is to find the smallest satisfying cardinalities for the abstracts by minimizing the sum of the union of the variables of all of the abstract sets.

3.1.3 Encoding `PROPERTIES`

Seeing as ultimately abstract sets will be replaced with concrete sets of selected sizes, for this encoding to be correct, it must depend on the sizes of sets in the interpretation and only be satisfiable if the original machine's `PROPERTIES` are satisfiable. Because of this, the focus in encoding the `PROPERTIES` is to reflect their constraints as comparisons of set cardinalities. This can require, on top of the already included abstract and concrete sets, new auxiliary sets. These auxiliary sets each get their own array of propositional variables to represent their potential elements, and differ from the concrete sets in not having, a priori,

a predefined size and from the abstract sets in that they are not necessarily non-empty. When the upper bound on abstract set sizes is k, and there is no other information about the intermediary set's size, it is k-dimensional. Sometimes an auxiliary set can be given a larger array if there is reason to do so, and because the upper bound is for abstract sets. For example, if the auxiliary set is known to represent a set with l more elements than a given abstract set, then the auxiliary set's array will be $k + l$-dimensional to allow for the possibility of the bound on the abstract set's size being tight.

Subsequently, the `PROPERTIES` constraints are encoded fundamentally as comparisons of set sizes. Simply this would come to affirming the difference between two sets being greater than a given integer according to the constraint in question, and this would look like:

$$\sum_{x \in ArrA} +1x + \sum_{y \in ArrB} -1y \geq \alpha \tag{3}$$

However, because of some auxiliary sets being defined relative to other sets and some of the abstract and concrete sets' cardinalities depending only indirectly on other sets' cardinalities, and because not every `PROPERTIES` constraint is simple and encodable in one constraint, but rather being constraints depending on others, the naive constraint schema in Inequality 3 is not always adequate.

Instead, again more auxiliary variables are introduced to represent constraints that are used for reasoning but not necessarily affirmed unconditionally, and that are designed to be true if and only if the constraint is true. This can be formalized, with s as the auxiliary variable, as:

$$s \Leftrightarrow \sum_{x \in ArrA} +1x + \sum_{y \in ArrB} -1y \geq \alpha \tag{4}$$

To ensure that this situation holds, all constraints of the type found in Inequality 3 are replaced with two related constraints that bind s to a constraint in both directions. This works by finding some reformulation of the original constraint and including s with a large enough coefficient, M, that render the other variables irrelevant for that constraint depending on the valuation of s. Each M is determined by the sets being compared and the degree of their comparison. Because of this, the M can differ for every reifying auxiliary variable, and even between the two directions of implication that are required to bind the variable to the original constraint.

This is most straightforward when s implies the constraint. After, in order to abide by the restricted format, avoiding negative literals by inserting s' integral complement in one instead of $\neg s$, the entailment of the original constraint by s is assured by such inequalities as:

$$M(1 - s) + \sum_{x \in ArrA} +1x + \sum_{y \in ArrB} -1y \geq \alpha \tag{5}$$

The required M needs to be large enough for the constraint to be satisfied whenever s, the antecedent variable, is false. Such a sufficiently large number

can be selected by starting from the degree, and adding the number of variables that count as potential elements of the subtrahend set, that is, the degree plus the size of the array of variables representing the second set or $\alpha + |ArrB|$. This would allow for the constraint to be made true once s is false, even if the first set takes its lowest possible cardinality, which is zero when any of abstract, concrete or, most notably, auxiliary sets are allowed for, and if the second set takes its greatest possible cardinality, which is the size of the array of variables that represents it. Plugging in this M expands to:

$$\alpha + |ArrB| - (\alpha + |ArrB|)s + \sum_{i=1}^{|ArrA|} +1x_i + \sum_{j=1}^{|ArrB|} -1y_j \geq \alpha \tag{6}$$

Then, after rearranging for tidiness we arrive finally at Inequality 9.

Conversely, enforcing the disentailment of the original constraint by the negation of s requires combining the negation of the original constraint with s multiplied by some M. The negation of the original constraint is as laid out as:

$$\sum_{x \in ArrA} +1x + \sum_{y \in ArrB} -1y < \alpha \tag{7}$$

Yet, to suit the restricted format, some adjustments are called for. Both sides are negated in order to reverse the comparison's direction and one is added to the smaller side to weaken the strict superiority. Continuing from there by positioning the newly positive sum of the second set's elements before the newly negated sum of the first set's elements, the prvious inequality is more felicitously formulated as:

$$\sum_{y \in ArrB} +1y + \sum_{x \in ArrA} -1x \geq 1 - \alpha \tag{8}$$

From this point in the derivation, one need only add s' positive literal multiplied by some M, to arrive at where Inequality 5 is in the other direction. The required M is once again the sum of the right-hand side of the inequality and the coefficient of the subtrahend, and there is not even need to tidy up the inequality thereafter, giving Inequality 10 as the result.

$$(-\alpha - |ArrB|)s + \sum_{i=1}^{|ArrA|} +1x_i + \sum_{j=1}^{|ArrB|} -1y_j \geq -|ArrB| \tag{9}$$

$$(|ArrA| + 1 - \alpha)s + \sum_{j=1}^{|ArrB|} +1y_j + \sum_{i=1}^{|ArrA|} -1x_i \geq 1 - \alpha \tag{10}$$

Equipped with these two inequalities, the `PROPERTIES` are explored recursively, passing reifying auxiliary variables to higher constraints. If the constraint is not depended upon by another constraint, but directly in the specification in its own right, then the selector is asserted to be true in the following way:

$$+1s \geq 1 \tag{11}$$

An equality would be correct as well, but it is nice for solvers to favour greater than or equal to when possible.

Small Example As a demonstration, let us take a small upper bound of $k = 4$ and consider parts of the beginning of the following machine encountered during the B MOOC[2]:

```
MACHINE
  Club
SETS
  REPORT = {yes, no};
  NAME
```

⋮

```
PROPERTIES
  capacity : NAT1 &
  capacity <= 100 &
  NAME : FIN(NAME) &
  card(NAME) > capacity &
  total : NAT1 & total > capacity &
  MAX_NAME : NAT1 &
  card(NAME) = MAX_NAME + 1
```

⋮

To the enumerated set `REPORT` are attributed the first variables, and, because the set is known to have exactly two elements, these are `x1` and `x2`, which must both be true. This gives the constraint:

```
+1 x1 +1 x2 >= 2;
```

The abstract set `NAME` could have any super-zero cardinality up to k, so receives the next four variables, at least one of which must be true:

```
+1 x3 +1 x4 +1 x5 +1 x6 >= 1;
```

Turning to the final constraint in `PROPERTIES`, one can read that the cardinality of `NAME` is one greater than `MAX_NAME`. `MAX_NAME` is first found in the `CONSTANTS` section, which is important for the specification of a B machine, but is significant to *set-sizing-for-counter-models* solely in so far as it occurs in `PROPERTIES`. When a human reads the machine, it is evident that `card(NAME) - MAX_NAME = 1` would suffice, but to facilitate reading POG files more autonomously, an intermediate set is introduced for `MAX_NAME + 1`.

[2] https://mooc.imd.ufrn.br/course/the-b-method

Because of previous sets and variables, `MAX_NAME` happens to be encoded by the four Boolean variables from `x33` to `x36`, and `(+)(1,MAX_NAME)` is encoded by the five Boolean variables from `x44` to `x48`, both times inclusive. The variable reifying the state of affairs where the new set is at least one greater than `MAX_NAME` is `x56`, and the variable reifying the fact that `MAX_NAME` is at most one less than the new array is `x57`. Both variables are encoded and asserted. The variable `x56` is encoded as follows, and `x57` is treated analogously.

```
-5 x56 +1 x44 +1 x45 +1 x46 +1 x47 +1 x48 -1 x33 -1 x34 -1 x35 -1 x36 >= -4;
+6 x56 +1 x33 +1 x34 +1 x35 +1 x36 -1 x44 -1 x45 -1 x46 -1 x47 -1 x48 >= 0;
+1 x56 >= 1;
```

The handling of the equality of `NAME` and `(+)(1,MAX_NAME)` is again similar.

Version 0
This version handles:

- both binary predicates,
- some expression comparisons,
- quantified expressions if they are existential,
- the unary predicate, and
- both n-ary predicates.

3.2 From OPB to POG

By running a pseudo-Boolean solver on the encoding from the previous subsection, the software looks for set sizes of abstracts sets in a machine with which the constraints set by the hypotheses can be satisfied. If definite sizes are found that fulfill these requirements and the goal fails to be satisfied, then the machine is known to be invalid, and the instantiated sets are themselves a counter-model to the machine.

This software helps get to the desired state by finding models of the premisses. As in Algorithm 1, *set-sizing-for-counter-models* explores the `SETS` and `PROPERTIES` sections of a POG file, and when the encoding from the previous subsection is ready, it is passed to a pseudo-Boolean solver that attempts to find a solution. The successful situation arises when the solver finds such a solution, which can then be used for setting concrete sizes in a new POG file. This will be returned if found. If not, either there is none to be found or the upper bound is too low to find one.

Algorithm 1 *set-sizing-for-counter-models*

pog ← input POG file
k ← upper bound on set size
enc ← *k*-parameterized, pseudo-Boolean encoding of *pog*'s sets and prerequisites
model ← model output by pseudo-Boolean solver run on *enc*
if *model* is not nil **then**
    outpog ← *pog* with abstract sets replaced with concrete sets of found sizes
    Return *outpog*
else
    Return nil
end if

4 Full Pipeline

set-sizing-for-counter-models accepts input in POG format, meaning the XML files translated to with the full `pogenerate` command (or `po`, both with argument `0` after the machine name) of ClearSy's Atelier B[3] from machines written in the B language.

If suitable sizes are found, the tool can output a POG file with the abstract sets replaced with appropriate concrete sets.

4.1 Bigger Picture

To integrate the outputs of *set-sizing-for-counter-models* into an SMT-solving tool, the concretized POG file can be translated into SMT-lib (or TPTP) format using ClearSy's pptranspog[4] tool. If the SMT problem instance is satisfiable, then the model it returns is a counter-model to the machine. Such a pipeline for this purpose can be read from Algorithm 2.

5 Experimentation

As another part of the BLaSST project, ClearSy has provided anonymized POG files from 5434 real-life B language machines defined for clients' purposes.

Unless there are mistakes, all of the machines were valid, and should never have counter-models. This would make *set-sizing-for-counter-models* irrelevant for any of the examples, so the data need to be preprocessed for experimentation. A machine is a miracle machine if its premisses are contradictory, and so will be trivially valid regardless of the quality of its goals. If a machine is originally valid, then, as long as it is not a miracle machine, negating its goals will lead to an invalid machine.

[3] https://www.atelierb.eu/en/atelier-b-tools/
[4] https://github.com/CLEARSY/pptranspog

Algorithm 2 Seeking Counter-Models

```
inpog ← input POG file
k ← upper bound on set size
newpog ← set-sizing-for-counter-models(inpog, k)
if newpog is not nil then
    smt ← pptranspog(newpog)
    state ← SMT-solver(smt)
    if state is sat then
        counter_model ← satisfying model
        Return counter_model
    end if
end if
Return failure
```

This negation was applied to all of the provided POG files. This application does not guarantee that each proof obligation will be invalid (because only one needs to be invalid to invalidate the machine), or that any will be (because it could have been a miracle machine), so the new generations had to be checked for validity by other means. In particular, two SMT-solvers, cvc5[5] and z3[6], were run on the pptranspog translations of the generations. When either solver returned `unsat`, the proof obligation was taken to be valid, and so was removed from the test suite. A solver returning `sat` could fortify credence in the proof obligation being invalid, but in fact this very rarely happened. Instead, answers of `unknown` or timeouts were common. This fact implies that there is not certainty about the invalidity of the proof obligations used in the experiments, which is unfortunate, but if the SMT-solvers had easily found invalidity on their own, such a result would undermine the value of *set-sizing-for-counter-models* in the first place.

For experimentation, *set-sizing-for-counter-models* was run on the negations that are not yet believed to be valid and that include abstract sets in their specification. These constraints reduce the usable machines from 5434 to 3809.

The experiments tested how efficacious and efficient the tool is when applied to the use case of helping churning out counter-models from POG files using SMT-solvers after SAT-based preprocessing.

1. First the POG files were input to *set-sizing-for-counter-models* with a maximum set size of 50 elements, which possibly output encodings as pseudo-Boolean optimization problem instances.

2. These optimization problem instances were input to Exact[7] with the function to minimize being the sum of all the variables representing possible elements of the originally abstract sets.

3. If Exact successfully produced a model, new POG files were generated

[5] https://github.com/cvc5/cvc5
[6] https://github.com/Z3Prover/z3
[7] https://gitlab.com/nonfiction-software/exact, [2]

Table 1: Summary of Results with Upper Bound of 50

	Number	Avg. time (ms)
Encoding as `.pbo`	231	10.2251
UNSAT PB	90	
Optimal models	141	47.994
To POG	141	6.42553
To SMT-lib	141	29.5035
With cvc5	104	274062
With Z3	113	319.28
Successes	117	137285

that were instances of the original machine substituting all abstract sets by small concrete sets that were still large enough to satisfy the `PROPERTIES`.

4. Finally, these new generations were translated to SMT-lib with pptranspog, before the same two SMT-solvers as earlier were run on them. Specifically, the exact calls for each were:

   - `cvc5 --incremental --mbqi --tlimit-per 180000 --stats <smt2_file>` and
   - `z3 -t:180000 -st <smt2_file>`.

5.1 Results

Table 1 contains the results from running these steps as the software currently stands. Of the 3809 POG files run on, 231 are successfully encoded to the OPB format. Obstacles to the others can be the software finding already that the maximum set size is insufficient and gives up, or it encounters operations in the `PROPERTIES` that it is not confident about handling. It is important that the satisfiability of output encodings be sound, so only if it is guaranteed that the constraints be as strong as in the original specification can an output be trusted. Some operations will always be risky, such as those related to universal quantification, but others may just not yet be implemented. There were no timeouts during this part of the experiment.

More than half of the SAT-encodings were optimally solved by Exact. Another solver, minisat+[8], did not manage to decide as many instances within the time limits. Another possibility for a solver is to return a model without guaranteeing that it optimizes function, but this did not happen for Exact. The final possiblity for a solver is to return `unsat`, which Exact did 90 times, and this means that the supremum may need to be increased for those instances.

[8] https://github.com/niklasso/minisatp, [1]

Table 2: SMT Solving by Proof Obligation

	Number	Avg. Time (ms)	Best Time (ms)
PO	3255 (23.0851)		
With cvc5	104 (505,2515,235)	274062	7654.54
With Z3	113 (348,2528,379)	319.2889	8.9936

In fact, 50 of these were due to a set being equated to the set of natural numbers or integers, which would be out of our reach. For smaller numbers, it is important to find a balance between a high upper bound which runs less risk of forcing `unsat`, and a low upper bound which makes the search space smaller for the solver. If an average of the headers of these 231 encodings is taken, it comes to `* #variable= 4892.4 #constraint= 5340.61 #equal= 7.4632 #intsize= 8.23377` (although, of course, each value is natural in actuality).

The row after the ones concerning pseudo-Boolean solution regards the time taken to read the sizes of the abstract sets from the output model and to replace these sets appropriately in the POG file. The following row deals with pptranspog's times for translating to the SMT-lib format. Overall, this results in a not very long expected time of preprocessing before running an SMT-solver.

The next rows give the average time spent by SMT solvers *per POG file*. The times make the SMT solution the longest part of the pipeline, and especially so when the solver used is cvc5. Table 2 gives a finer grained overview of the performance of the solvers. The first row shows the count of all proof obligations shared among the 231 POG files, with the average number of proof obligations per POG file in parentheses. The following rows again compare the SMT solvers but include information about how many individual proof obligations are found `sat`, `unsat` or `unknown` in parentheses. The average time is, as in Table 1, for whole POG files, whereas the best time is the average time for at least one proof obligation in a file to be found `sat`.

5.2 Discussion

5.2.1 Runtimes

The average time for getting to a counter-model is prohibitively long for most applications of a counter-model finder. This is due to the runtimes of SMT solvers.

The time for SMT solution in Table 1 is considering whole POG files, which can have several goals. Each goal is checked separately in SMT, and although one goal may have been found `sat`, invalidating the whole machine, the solver continues to check each subsequent goal, even when there are many and some are difficult. The runtime could be shortened greatly (and, in practice, would *have* to be shortened) by returning the first model found instead.

Table 2 gives a clearer notion of how performance would be if POG files were

not read incrementally, but rather were first split into proof obligations, then all of these proof obligations were run on in parallel, and the tool returned a model as soon as one were found for any of the proof obligations. Because verifying the machine requires all of the proof obligations to be proved, it would suffice to find a counter-example to one of them.

Even when considered proof obligation by proof obligation, cvc5 gives a significantly longer average best time than Z3. Comparing the number of `sat` proof obligations, cvc5 succeeds to finding models to more hard instances, and its average times to success are heavily influenced by the difficulty of these instances. Nevertheless, there are only four POG files for which cvc5 finds some counter-example but Z3 does not. There are several possibilities about how it would be best to organize the solvers: maybe Z3 would be sufficient alone, maybe both should be run as a portfolio, maybe it would be best to know in advance how difficult a problem instance is and to choose a solver accordingly, maybe something else.

Although the mean times are quite high with respect to the rest of the tool chain, very often the times are much shorter even for the SMT solvers.

5.2.2 Specific Notes about SMT Translation and Solution

The translation to SMT-lib does not use the currently public version of pptrans-pog. When using the version available on GitHub, both solvers are unable to respond `sat` for any of the instances, and so cannot produce counter-models. Instead, a version that is currently under development was used, and the results are much improved. One can expect that in the future such functionality will be shared with all.

Some of the times that cvc5 reported unknown, it still output a model. These models were instantiations of the constraints without accounting for quantification. It could arise that these are sufficient, depending on how quantification occurs in the machine, so could be an aspect to look into for improvement. These tentative models cannot be trusted as easily as the not-necessarily-optimal models of pseudo-Boolean solvers, but they may be a starting point for further exploration.

6 Final Remark to Remember

It is important to consider limits to the tool. If, even with the maximal size of set considered, the tool does not find the premisses to be satisfiable, it does not mean that the proof obligation is trivially valid, but rather that the hypotheses cannot be satisfied with sizes below the maximum considered. It could still be the case that they are satisfiable with greater sizes.

On the other hand, when the tool finds suitable set sizes and the goal is found to be valid with respect to those sizes, it is not at all an informative situation for verification. Because only one of the possible combinations of sizes is considered, it does not rule out the refutability of the goal in other interpretations.

The only interesting result is when sizes are found that can satisfy the premisses while the goal can still be falsified.

7 Bibliographic References

[1] Niklas Eén and Niklas Sörensson. "Translating pseudo-boolean constraints into SAT". In: *Journal on Satisfiability, Boolean Modeling and Computation* 2.1-4 (2006), pp. 1–26.

[2] Jan Elffers and Jakob Nordström. "Divide and Conquer: Towards Faster Pseudo-Boolean Solving." In: *IJCAI*. Vol. 18. 2018, pp. 1291–1299.

[3] Olivier Roussel. *Restricted OPB Format in Use in the PB Competitions*. 2024. URL: `https://www.cril.univ-artois.fr/PB24/OPBcompetition.pdf`.

[4] Olivier Roussel and Vasco Manquinho. "Pseudo-Boolean and cardinality constraints". In: *Handbook of satisfiability*. IOS Press, 2021, pp. 1087–1129.