CPMpy submissions to the PB26 competition

Tias Guns
Department of Computer Science
KU Leuven
Leuven, Belgium
tias.guns@kuleuven.be

Thomas Sergeys
Department of Computer Science
KU Leuven
Leuven, Belgium
thomas.sergeys@kuleuven.be

Ignace Bleukx
Department of Computer Science
KU Leuven
Leuven, Belgium
ignace.bleukx@kuleuven.be

Hendrik Bierlee
Department of Computer Science
KU Leuven
Leuven, Belgium
henk.bierlee@kuleuven.be

***Abstract*—CPMpy [6] is an open-source Python library for modeling combinatorial (optimisation) problems. Solvers from different CO paradigms support different types of decision variables and constraints. Through a series of equivalence-preserving constraint reformulations, CPMpy can transform a constraint model into the target format, enabling systematic comparisons of solvers across paradigms. For this competition, we use CPMpy as a middleware that can translate *.opb* input to the solver APIs of varying technologies: SMT, CP, (M)ILP, PB, and (Max)SAT. Included in this submission are LCG CP solver OR-Tools CP-SAT, ILP solver HiGHS, and PB solver Pindakaas-CaDiCaL. The purpose of our submission is to widen the range of solving technologies evaluated in the competition, while reusing and stress-testing the transformation capabilities of CPMpy.**

Code : CPMpy's code is available on GitHub: https://github.com/CPMpy/cpmpy
Acknowledgement : This research received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant No. 101002802, CHAT-Opt) and Proof of Concept (POC) grant (Grant No.101158376, PO-CPMpy).

I. Format loaders

In the recent work of [10], we introduced a collection of file format readers and loaders to CPMpy. These tools convert well-known benchmarking instances into CPMpy models, giving access to our constraint reformulation stack; an extensive set of rewrite operations such as constraint decomposition, linearization, and Boolean encodings, that allow transforming between different constraint formalisms.

II. Constraint reformulations

Solvers from different CO paradigms support different types of decision variables and constraints. Constraint Programming (CP) has the most expressive language, supporting global constraints [12], potentially non-linear, over both integer and Boolean decision variables. Next are Integer Linear Programming (ILP) solvers supporting only linear inequality constraints, Pseudo-Boolean (PB) solvers supporting linear inequality constraints over Boolean variables, and (Max)SAT-solvers which support Boolean formulas in Conjunctive Normal Form (CNF).

Solvers from the more expressive paradigms can take less expressive languages as input, but the reverse direction requires rewriting a set of high-level expressions into a set of lower level expressions, often requiring the use of auxiliary variables. This is achieved through a series of equivalence-preserving constraint reformulations, transforming a constraint model into the target format. We identify five main reformulation stages: decomposition, flattening, linearization, encoding integer variables to Booleans, and encoding pseudo-Boolean constraints into CNF. Each of the transformations is implemented in CPMpy as a separate function.

A. Decomposition

When a CP solver does not support a particular CP *global constraint*, or when targeting a lower level paradigm, that global constraint needs to be rewritten into a collection of equivalent lower level expressions, i.e. decomposing the constraint. Examples of global constraints include AllDifferent, Element, Cumulative and Circuit.

We partition the constraints resulting from the decomposition into two groups; *value* and *defining*. The *value* constraints represent the logical value of the original constraint. When a global constraint occurs in a reified or nested context (e.g., under implication, disjunction, or negation), these constraints must themselves be reified or nested to preserve compositional semantics [1]. *Defining* constraints define the value of new auxiliary variables introduced by the decomposition, and should always be enforced at top-level. Returning both groups of constraints separately prevents unnecessary reification or nesting. In practice, CPMpy also uses the concept of global constraints to decompose non-linear functions such as Max, Min, Multiplication and Division.

B. Flattening

The second step in the transformation pipeline is to eliminate *nested* expressions; introducing auxiliary variables such

that each constraint is over decision variables rather than over other expressions. To improve the output of this process, we reuse auxiliary variables through common subexpression elimination (CSE). Whenever we introduce an auxiliary variable for a sub-expression, we first check if such a variable already exists in a cache we maintain.

C. Linearization

To transform the constraint model into a linear or Boolean model, we linearize the global constraints and functions. We employ standard decompositions for most global constraints available in the CPMpy library, and use a few alternative linear-friendly decompositions for some often-used global constraints. These decompositions encode the integer variables into auxiliary Boolean variables by using a direct encoding. For example, $\text{ALLDIFFERENT}(X)$ is decomposed into $\sum_{x_i \in X} [\![x_i = v]\!] \leq 1$ for each value in the domain of the variables in X. Similarly, $\text{ELEMENT}(A, idx, v)$ is decomposed to $\sum_{i=1..|A|} a_i [\![idx = i]\!] = v$.

Additionally, we detect the use of categorical variables and encode them once using a direct encoding, rather than linearizing each $b \Leftrightarrow (x = v)$ expression separately. This is done by checking whether comparison constraints such as $(x = v)$ are used as a subexpression. E.g., the constraint $(x = 3) \vee (y \neq 1)$ is linearized by first adding the direct encoding of both x and y and then linearizing the constraint as $[\![x = 3]\!] + \neg[\![y = 1]\!] \geq 1$, with $[\![x = v]\!]$ the Boolean variable from the direct Boolean encoding.

D. Boolean encoding of integer variables

To encode Constraint Programming models or ILP models for (Pseudo-)Boolean solvers, all integer variables must be encoded into Boolean variables. We currently support the most common encodings: direct encoding [13], order encoding [11] or log-encoding (also known as binary encoding) [5]. We reuse the direct encoding if it already exists, and otherwise heuristically decide an encoding based on the size of the domain.

CNF encoding of pseudo-Boolean constraints: When targeting a constraint model for (Max)SAT, the last stage in our transformation pipeline consists of encoding all pseudo-Boolean constraints into clausal form. For this final step, we connect to standard tools from the (Max)SAT community, namely PySAT (including pypblib) [8] and the pindakaas library [3], both of which contain a large number of state-of-the-art CNF encodings for pseudo-Boolean constraints.

All the above transformations include encoding choices, e.g. how to decompose the different global constraints, what integer encoding to use, what PB encoding to use etc. These choices are still under active study in the community today to determine different trade-offs related the size of the encoding and the overall runtime performance of the solver. Any library that aims to transform between different formalisms will have to set some reasonable default choices, these can always be changed or improved by expert users.

III. Solvers

CPMpy can translate to the Python API of many different solvers spanning CP, MIP, SMT, pseudo-Boolean and SAT technologies. Our CPMpy submissions bundle the following solvers:

- `cpmpy_ortools` [9] (version: 9.15.6755) The default solver in CPMpy, it is a lazy clause generation CP solver. It supports parallel search too.
- `cpmpy_highs` [7] (version: 1.14.0) An high performance serial and parallel linear optimisation solver, supporting LP, QP and MIP.
- `cpmpy_pindakaas_cadical` [4] [2] (version: 0.4.1) Pindakaas is a Rust library to transform pseudo-Boolean and integer constraints into CNF, using CaDiCal as the default solver backend.

We did not significantly contribute to the development of any of these solvers, with the exception of the Pindakaas library, and are grateful to the solver developers for making their technology available.

References

[1] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. *Constraints An Int. J.*, 18(1):1–6, 2013.

[2] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

[3] Hendrik Bierlee and Jip J. Dekker. Pindakaas (pyndakaas-v0.4.1), 2026. doi:10.5281/zenodo.10851855.

[4] Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In *IJCAI*, volume 18, pages 1291–1299, 2018. URL: https://gitlab.com/JoD/exact.

[5] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic sat-compilation of planning problems. In *IJCAI*, pages 1169–1177. Morgan Kaufmann, 1997.

[6] Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation at CP (Modref 2019)*, volume 19, 2019.

[7] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL: https://www.gurobi.com.

[8] Alexey Ignatiev, António Morgado, and João Marques-Silva. Pysat: A python toolkit for prototyping with SAT oracles. In *SAT*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018.

[9] Laurent Perron and Frédéric Didier. Cp-sat. URL: https://developers.google.com/optimization/cp/cp_solver/.

[10] Thomas Sergeys, Ignace Bleukx, and Tias Guns. Unified programmatic access to co benchmarks, to connect constraint solving communities. In *Proceedings of the 29th International Conference on Theory and Applications of Satisfiability Testing (SAT 2026)*, 2026. Accepted for publication.

[11] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints An Int. J.*, 14(2):254–272, 2009.

[12] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006. doi:10.1016/S1574-6526(06)80010-6.

[13] Toby Walsh. SAT v CSP. In *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.