Increasing modeling language convenience with a universal n-dimensional array, CPpy as python-embedded example

Tias Guns

July 19, 2019

Abstract

CP modeling languages offer convenience to the user by allowing both constants and decision variables to be first class citizens over which mathematical and Boolean operators can be expressed. Furthermore, text-based modeling language are often solver-independent: they parse the textual model and generate an expression tree of constraints and variables, over which solver-specific transformations can be applied.

We demonstrate that the convenience of modeling languages can be further increased by taking inspiration from the wider scientific computing field. More specifically the use of n-dimensional arrays, sometimes called tensors, as universal data structure. In this setting, both constants and n-dimensional arrays are first class citizens over which operators can be expressed. In the latter case these are called vectorized operations, for which highly efficient implementations in low-level languages different from the host language can be used.

In this short paper, we argue that the convenience of CP modeling languages can be further increased by embracing constants, decision variables and n-dimensional arrays as first-class citizens. The vectorized operations over the n-dimensional arrays allow for more compact models by using advanced indexing. In fact, implementing advanced indexing in text-based modeling language may require substantial effort while in an embedded modeling language this can be offloaded to existing implementations such as the widely popular *ndarray* of NumPy. We demonstrate this in a python-embedded CP modeling language that we call CPpy. To ease reasoning and transforming the resulting expression tree of constraints, we also define a minimal class diagram of expressions over decision variables, that is as oblivious to the actual constraint reasoning as possible.

1 Introduction

Modeling languages are an integral part of the constraint programming community, and have both a wide uptake in their use and an active research community. Examples of text-based modeling languages are MiniZinc [13], Essence [7], OPL [15]. Their key properties are modeling convenience and (except for OPL) solver-independence.

The alternative to text-based modeling languages are modeling languages embedded in an existing programming language. All constraint solvers offer an API to construct models, and hence could be considered modeling languages. However, they are often highly tied to the underlying solver implementation and take less care to provide modeling conveniences where both constants and decision variables are equal first-class citizens.

An exception to the above is Numberjack [10], a python-embedded modelling package that was introduced almost 10 years ago. It builds a solver-independent expression list, as do text-based modeling languages, but by operator overloading instead of parsing. It supports solver-specific constraint decomposition and can translate models to the Mistral CP solver, to the MiniSat SAT solver and to the SCIP MIP solver.

Our work takes inspiration from Numberjack, but further aims to use n-dimensional arrays as first class citizen. In fact, Numberjack already includes its own 'Matrix' object to allow more convenient indexing into a two-dimensional (decision variable) array. However, this is a limited implementation and unneeded when embracing existing n-dimensional array packages such as NumPy.

NumPy is a popular package for scientific computing in Python. It's n-dimensional array object is called *ndarray*. It has highly efficient low-level implementations for storage and vectorized operations. It further allows for advanced indexing with different ways of specifying which elements to select, including logical expressions on other ndarrays. This contributes to its popularity for doing numeric computations and data analytics tasks.

N-dimensional arrays have already been in use in scientific programming languages like Matlab and R for many years. NumPy has allowed for similar use in the more general purpose programming language Python. Other Python packages are built on top of NumPy, including SciPy which contains linear programming routines, and CVXpy which is a general convex optimisation framework. Popular AI frameworks such as the deep learning tools TensorFlow and PyTorch are fully compatible with NumPy's ndarrays as well, and often require the input to be in that format.

This paper explores the possibilities of using the full power of n-dimensional arrays and vectorized computations, without having to re-implement these capabilities in a text-based language compiler. In fact, the resulting language and expression tree generator aims to be as lightweight as possible so that all model transformations can be offloaded to existing solver-independent frameworks that already do that very well, such as Numberjack [10], MiniZinc [13] and XCSP3 [3]. There is still much value in such an embedded language, is it is compatible with the Python family of scientific tools allowing both easy integration which such tools for CP practitioners, and potentially easier uptake of CP technology by practitioners of other scientific Python packages.

2 Modeling languages

Properties of text-based modeling languages Apart from OPL, the text-based modeling languages are typically also solver independent. By also being programming language independent, they are considered user-friendly and easy to learn, as one need not be versed in a specific programming language and the syntax of the modeling language is typically kept high-level and mathematical-like, which is intuitive to modellers.

However, there is still syntax to learn, especially for more complex constructs such as dealing with matrices, doing array comprehension or pretty printing. Furthermore, conveniences available in modern programming languages may not be available in the modeling languages, including array slicing, data manipulation, debugging and printing/catching errors during model creation. The reason is often simply a matter of priorities and required time on the language developer side: these are not essential to a constraint modeling language and can even complicate the compiler implementation.

To ensure extensibility, modeling languages like MiniZinc allow user-defined functions as well as custom global decomposition definitions that are solver-specific. These can be written in the syntax of the modeling language, allowing easy extensions of these kinds.

However, any kind of extension or model transformation that requires access to the expression tree has to be added to the compiler. And this compiler is written in a different language than the modeling language, which can be a serious barrier to entry. In the case of MiniZinc, this is sometimes worked around by parsing its flattened FlatZinc output and further transforming that structure (as done in MiningZinc [8] using multiple global definitions), or even reconstructing the original structure and further transforming that (as done by or-tools and non-CP solvers).

Moreover, uses of CP where the solver is called multiple times with minor variations to the model, such as basic large neighborhood search [5] or dominance programming [9] require either extending the language with search constructs, and hence the compiler which may not support states and nesting yet as it is not needed for modeling, or to do this outside of the modeling language. In the latter case, a (inherently different) programming language must be used and

communication with the solver has to happen by writing and parsing text-based models and solutions which can be cumbersome.

Extensive experiments with different datafiles and solvers also require a programming language in which the compiler/solvers is called multiple times from the command line, with different text files an input parameters. This is no different from the use of a text-based problem formats such as DIMACS CNF, MPS/LP and XCSP, however these typically do not include convenience modeling constructs and largely require the use of a programming language to generate a valid problem file to start with.

Properties of embedded modeling languages The advantage of being embedded in a programming language is that the whole pipeline of operations, of which solving is one, can be done within the same language: data reading and prepocessing, model construction, search specification, solving and iterative solving, debugging and printing, visualisation and post-processing of the solution(s).

The main disadvantage of an embedded modeling language is that it is tied to one specific programming language, which the user must be familiar with or learn, and where the IDE tools of that language must be used and familiarized with. In case of languages like Java and C++, this often involves additional code constructs (boilerplate code) that can increase the initial learning curve. However, modern languages like Python and Julia have almost no required boilerplate and look more similar to text-based modeling languages even if they are full-fledged programming languages.

It seems that the language constructs used to build a CP model within a programming language, is usually not called a modeling language. Still, it has all the constructs of a text-based modeling language (variables, operators, constraints) that can be composed in predefined ways, and it typically does transformations on these constructs prior to solving. Perhaps the reason is that modeling language implies a form of solver independence, and in embedded languages usually the constructs are closely intertwined with one specific solver and its internal representation.

Embedded solver-independent languages We are aware of one embedded modeling language that is solver independent, namely Numberjack [10]. Almost 10 years ago, the authors already argued for the advantages of being both embedded in a popular language and supporting multiple combinatorial solvers: reuse of existing language constructs, extensibility of (decomposable) constraints and search strategies in the same language and integration with other packages of that language such as spreadsheet readers, web-based GUIs and automatic cloud deployment.

Further use cases that have emerged in recent years: iterative solving [9], natural-language explanations of CP inference [14], calling a solver from within a machine learning procedure [6], feeding machine learning trained networks [2] and graphs [4] into solvers and interleaving optimisation and simulation [1]. It also makes it easier to benefit from developments in other AI fields, for example the use of generic algorithm configuration methods like SMAC [11] which is available as a Python package [1].

Compared to Numberjack, we propose to go one step further by 1) embracing n-dimensional arrays as universal data structure as explained earlier, and 2) keeping the abstract syntax tree representation as unaware of solving as possible.

The motivation for having abstract expressions is two-fold: firstly, all model transformation code (propagation of constants, simple rewritings, the use of variable views, decompositions) can be kept separate and re-used, or not, for different purposes. Secondly, the resulting expression representation will be fairly simple and hence it can lower the barrier to writing new model transformation tools, either directly on the high-level model or on a previously transformed model.

[1] https://github.com/automl/SMAC3

3 CPpy

CPpy is a prototype python-embedded solver-independent language with the following **design principles**:

1. use a native n-dimensional array implementation as data-structure for variable arrays (in case of Python: numpy's ndarray)
2. use as much operators and syntax from the native language as possible, hence introduce as few custom constructs as possible
3. be solver independent, allowing both native solvers and text-based (meta-)solvers to be used as backend
4. keep as much constraint logic outside of the abstract syntax tree representation as possible
5. provide convenient access to the solution, directly from the variable objects created during modeling

3.1 Modeling language constructs

A single construct is offered to instantiate decision variables: BoolVar(shape) and IntVar(lb,ub, shape), where shape is the shape of the n-dimensional array, e.g. 1, 10 or (5,5) for a single element, array of 10 elements and 5x5 matrix respectively. Shapes of higher dimension are also possible. The underlying object will inherit from NumPy's *ndarray* and hence fully compatible with all numpy operations.

Python's operator overloading is such that the existing mathematical and Boolean operator syntax can be used on these ndarrays of variables. Our implementation of these overloaded operators then construct the relevant Expression objects (see next section) on the fly. Operations on constants are automatically performed by Python itself, only when an operation on a decision variable (or array thereof) is called will our code be called.

The only further convenience functions are: implies(left , right) which has no corresponding built-in operator in python, the minimize() and maximize() functions that create an 'Objective' instance as well as a function for every global constraint to create a corresponding 'GlobalConstraint' instance. We also had to overwrite the built-in Python functions all() and any() so as to construct the corresponding BoolOperators for n-ary 'and' and 'or'.

3.2 Examples

We now review two examples expressed in CPpy, that demonstrate that the syntax is of similar light weight as text-based modeling languages, and the benefits using n-dimensional arrays and vectorized operations.

Example 1: Send More Money This example demonstrates that apart from variable creation and global constraints like alldifferent(), the language's standard operators can be used, including built-in functions like sum(), '+' and '=='. Also note the direct access to variable values after solving, through the '.value' property.

Furthermore, it demonstrates how we can make use of NumPy also for convience of operations on constants, e.g. mathematical transformations on arrays (arange(n), which creates an array 0..n-1, the listwise power operator '$**$' and flip). It is unlikely that text-based modeling languages will support such constructs unless it frequently improves modeling convenience.

Send More Money in CPpy

```
from cppy import *
import numpy as np

# Construct the model
s,e,n,d,m,o,r,y = IntVar(0,9, 8)

constraint = []
constraint += [ alldifferent([s,e,n,d,m,o,r,y]) ]
constraint += [    sum(   [s,e,n,d] * np.flip(10**np.arange(4)) )
                 + sum(   [m,o,r,e] * np.flip(10**np.arange(4)) )
                == sum( [m,o,n,e,y] * np.flip(10**np.arange(5)) ) ]
constraint += [ s > 0, m > 0 ]

model = Model(constraint)
stats = model.solve()
print("  S,E,N,D =  ", [x.value for x in [s,e,n,d]])
print("  M,O,R,E =  ", [x.value for x in [m,o,r,e]])
print("M,O,N,E,Y =", [x.value for x in [m,o,n,e,y]])
```

Sudoku in CPpy

```
x = 0 # cells whose value we seek
n = 9 # matrix size
given = numpy.array([
    [x, x, x,  2, x, 5,  x, x, x],
    [x, 9, x,  x, x, x,  7, 3, x],
    [x, x, 2,  x, x, 9,  x, 6, x],

    [2, x, x,  x, x, x,  4, x, 9],
    [x, x, x,  x, 7, x,  x, x, x],
    [6, x, 9,  x, x, x,  x, x, 1],

    [x, 8, x,  4, x, x,  1, x, x],
    [x, 6, 3,  x, x, x,  x, 8, x],
    [x, x, x,  6, x, 8,  x, x, x]])


# Variables
puzzle = IntVar(1, n, shape=given.shape)

constraint = []
# constraints on rows and columns
constraint += [ alldifferent(row) for row in puzzle ]
constraint += [ alldifferent(col) for col in puzzle.T ]

# constraint on blocks
for i in range(0,n,3):
    for j in range(0,n,3):
        constraint += [ alldifferent(puzzle[i:i+3, j:j+3]) ]

# constraints on values
constraint += [ puzzle[given>0] == given[given>0] ]

model = Model(constraint)
stats = model.solve()
```

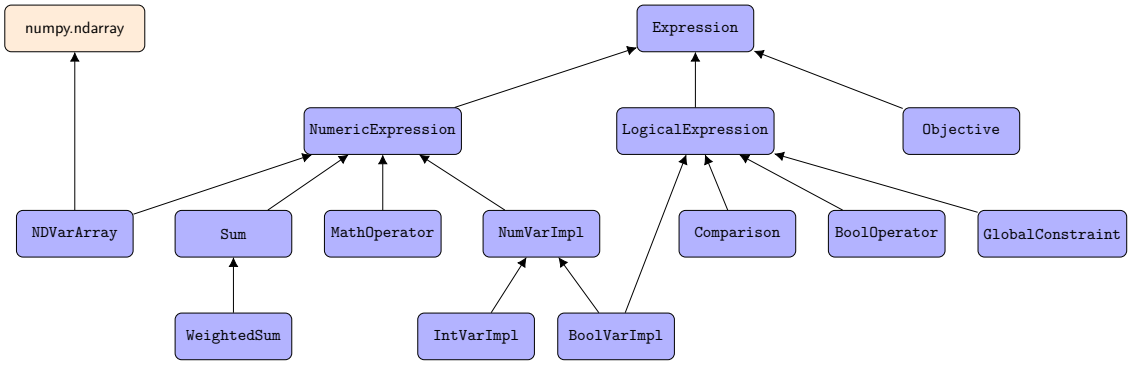
Example 2: sudoku The classical sudoku problem greatly demonstrates how standard operators over n-dimensional arrays like transpose and advanced iterators can also increase modeling convenience of CP problems!

There are two advanced iterators in this example, the first is 'i:i+3' which returns array [i,i+1,i+2] and can also easily be supported in text-based languages. The second is more subtle but also more powerful, namely 'given>0' in the last constraint. It returns the list of indices of all cells in 'given' whose value is greater than 0. This list of indices can be used as an index, both on the same n-dimensional array or on another (in this case the variable ndarray *puzzle*), such that only the elements at those indices are returned.

Furthermore, all the conversions from iterators and views on ndarrays to lists of variables are automatically performed by the ndarray implementation requiring no custom implementation.

3.3 Class diagram of expressions

A model is a list of expressions. Constraints and decision variables and all operators are types of expressions that inherit from a base Expression class. We only need to consider operators and constraints involving decision variables, all operations on constants will be immediately performed by the host language. The following is the complete class diagram of expressions:



We point out that we make convenient use of multiple inheritance for NDVarArray and BoolVarImpl. This limits the number of places where we need to implement operator overloading. In fact, this is mostly only in the NumericExpression and LogicalExpression classes. Python nicely supports multiple inheritance including automatic linearisation of superclasses to avoid problems with diamond inheritance patterns such as BoolVarImpl's.

Comparisons (==, ! =, >=, etc) and Boolean operators (and, not, xor, implied) do not have one class per operator, neither do mathematical operators. This is possible because it is an explicit design decision not to put constraint solving logic in the expression representation. Hence, there is no need for subclassing to the specific operators of the same family.

The only exception is Sum, the n-ary sum expression, and WeightedSum, its weighted variant. Sums of multiplications of a variable and a constant are automatically transformed into a weighted sum too. Weighted sums are less typical in programming languages to have their own construct, but very typical in CP. Exactly for the reason of detecting WeightedSums during construction did we decide to have explicit an Sum class too. Substraction is part of MathOperator, unless it is part of a bigger chain of addition/substraction (which is automatically detected), in which case it is turned into a single WeightedSum.

4 Discussion

The above examples show some of the potential benefits of using n-dimensional arrays and vectorized operations in constraint modeling languages.

We chose to create an embedded modeling language because this allows to reuse existing n-dimensional arrays, and eases interfacing to other packages in that language. The downside of the embedded language is that there is less freedom in deciding the syntax; in this case we can not define new keywords (only functions and operator overloading), and there is no operator symbol for implication $\rightarrow$ which is common in text-based constraint modeling languages.

To reach its full potential, the CPpy language should be linked to existing solver-independent solving frameworks including MiniZinc, Essence and XCSP3. Python-based interfaces to solvers including the Numberjack framework (and through that SAT solvers and MIP solvers) and or-tools' lazy clause generation solver should be included as well.

Furthermore, it would be interesting to be able to include a pure-Python solver, such as a Python version of MiniCP [12]. This would provide a complete framework where everything from solver-independent exploration to solver-specific extensions could be accomplished in the same language. From a pedagogical point of view, this could also align with the growing trend to already use Python for basic programming and data science courses.

The advanced possibilities of embedded solver-independent modeling languages obviously does not preclude the interest in text-based modeling languages. In fact, one of their key strengths are the compilers behind them, and their ability to tailor models to specific solvers. The ability to use such compilers as libraries, where the input is not a text file but an expression tree, would allow the same compiler to be used for both models resulting from parsing text files and from a language-embedded formulation. This could strengthen both approaches at once.

Code The code is open source and available online: `https://github.com/tias/cppy`

References

[1] Hamid Allaoui and Abdelhakim Artiba. Integrating simulation and optimization to schedule a hybrid flow shop with maintenance constraints. *Computers & Industrial Engineering*, 47(4):431–450, 2004.

[2] Andrea Bartolini, Michele Lombardi, Michela Milano, and Luca Benini. Neuron constraints to model complex real-world problems. In *International Conference on Principles and Practice of Constraint Programming*, pages 115–129. Springer, 2011.

[3] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: An integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398*, 2016.

[4] Rocsildes Canoy and Tias Guns. Vehicle routing by learning from historical solutions. In *International Conference on Principles and Practice of Constraint Programming*, pages 1–16. Springer, 2019.

[5] Jip J Dekker, Maria Garcia de la Banda, Andreas Schutt, Peter J Stuckey, and Guido Tack. Solver-independent large neighbourhood search. In *International Conference on Principles and Practice of Constraint Programming*, pages 81–98. Springer, 2018.

[6] Adam N Elmachtoub and Paul Grigas. Smart" predict, then optimize". *arXiv preprint arXiv:1710.08005*, 2017.

[7] Alan Frisch, Warwick Harvey, Christopher Jefferson, Bernadette M. Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[8] Tias Guns, Anton Dries, Siegfried Nijssen, Guido Tack, and Luc De Raedt. Miningzinc: A declarative framework for constraint-based mining. *Artificial Intelligence*, 244:6–29, 2017.

[9] Tias Guns, Peter J Stuckey, and Guido Tack. Solution dominance over constraint satisfaction problems. In *ModRef 2018 workshop on Modeling and Reformulation*. CP18 workshop, 2018.

[10] Emmanuel Hebrard, Eoin OMahony, and Barry OSullivan. Constraint programming and combinatorial optimisation in numberjack. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 181–185. Springer, 2010.

[11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[12] Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. Minicp: a lightweight solver for constraint programming (2018).

[13] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[14] Mohammed H Sqalli and Eugene C Freuder. Inference-based constraint satisfaction supports explanation. In *AAAI/IAAI, Vol. 1*, pages 318–325, 1996.

[15] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, 1999.