Coopy: Object Oriented Constraint Programming for Python

Adrián Barreal

Fundación Dr. Manuel Sadosky, Argentina
abarreal@fundacionsadosky.org.ar

Abstract

This paper describes Coopy, a Python library that dotes the language with seamless support for object oriented constraint programming capabilities. The Coopy approach is compatible with standard object oriented practices, allowing to conceive complex constraint models as systems of interacting objects.

1 Introduction

Constraint programming [12, Chapter 4] is a declarative programming paradigm in which the programmer does not provide a step by step procedure to find a solution for a problem, but instead describes properties about this solution in the form of constraints over a set of unknown variables. These constraints typically take the form of first order logic formulas. Once all desired constraints have been imposed, a constraint solver can be engaged to find an assignment of values for the unknowns such that all constraints are satisfied (if it exists). This programming approach has at least the following important benefits:

- The solution found is guaranteed to meet the constraints. Constraint programming, therefore, provides stronger guarantees than imperative programming practices.
- There are many problems for which stating the properties of the desired solution is much easier than writing an algorithm to find it. This makes constraint programming an appropriate tool to solve many otherwise difficult problems.

Object oriented programming, on the other hand, is the stadard practice in the industry of software development. It has been highly successful due to its modularity, reusability and reliability [5]. Object oriented programming allows to tackle large, complex programming tasks in a very efficient way, conceiving solutions as systems of interacting entities called objects, each one having attributes, behavior and state.

A widely adopted, easy to use and easy to learn object oriented language is Python. Python has become very popular in the scientific community, with projects such as TensorFlow [1], IPython [8], and SciPy [13], among others.

This paper describes Coopy, a Python library that relies heavily on operator overloading to dote the language with seamless support for constraint programming capabilities. The Coopy approach blends in well with the object oriented features of the language, allowing for the generation of complex constraint systems from object networks, instantiated from class models that can be still conceived and written as typically done in the standard object oriented practice. This differentiates Coopy from

other constraint programming packages, which are more procedural in nature and less transparent in their usage. The most distinctive feature of Coopy is its support for a seamless transition from constraint programming into imperative object oriented programming, allowing to use the former as an instantiation mechanism for the later. These reduces the amount of imperative code that needs to be written, and provides stronger guarantees about the properties of the instantiated objects.

The upcoming section 2 describes the basic concepts underlying the Coopy programming model. Section 3 discusses some applications, including bounded model checking, declarative object instantiation, and type synthesis from an axiomatic specification. Section 4 goes into more detail on what differentiates Coopy from other constraint programming packages. Some limitations of the approach are then discussed in section 5. Finally, the paper closes with a brief conclusion.

2 Basic Concepts

2.1 Symbolic Attributes

In the Coopy programming model, objects are allowed to have attributes that are *symbolic*. Symbolic attributes are the unknowns of the problem, the `x` of the equation. They are what the constraint solving process will yield values for. We may, for example, define a class that represents a colored node in a graph, in which the color of the node is represented by an unknown symbolic integer, as shown in the following code fragment:

```
from coopy import symbolic_int

class Node:
    def __init__(self):
        self._color = symbolic_int('c')
```

In this case, we say that `_color` is a symbolic attribute of the class `Node`. We may also call `_color` a symbolic variable. The concept and the terminology is essentially the same as that used in the field of symbolic execution and symbolic analysis of software [2][10].

Symbolic attributes have no concrete value, just a name and a type. In appearance, however, they behave just like regular Python primitives, and they can be used as such. The following code, for example, is perfectly valid:

```
x = symbolic_int('x')
y = symbolic_int('y')
z = 2*(x + y)
p = z > 5
```

The value of an expression depending on symbolic attributes is not backed by concrete Python primitives but by an abstract syntax tree (AST) instead. The AST encodes the whole history of operations, always as a function of the symbolic variables involved. In the above case, for example, the variable `p` would have a value described by the following tree:

$$p : (>) \begin{cases} z : (*) \begin{cases} 2 \\ (+) \begin{cases} x \\ y \end{cases} \end{cases} \\ 5 \end{cases}$$

This AST can at any point be compiled into a constraint for the constraint solver engine. This abstraction over the constraint building process is the one also adopted by symbolic analysis frameworks such as angr [11], symbolic computer algebra systems such as SymPy [6], and constraint solver engine interfaces such as that of the automated theorem prover Z3 [3], which is incidentally the one that Coopy relies on for its constraint solving tasks.

2.2 Constraints

Logical operations involving symbolic variables return not booleans but instances of `Predicate` objects instead, which essentially represent constraints. Consider, for example, the following extension to our `Node` class:

```
class Node:

    def __init__(self):
        self._color = symbolic_int('c')

    @property
    def color(self):
        return self._color

    def same_color_as(self, other):
        return self.color == other.color
```

In the following code fragment, the variable `q` will end up being a reference to a `Predicate` object of the concrete type `Equal`:

```
n1 = Node()
n2 = Node()

q = n1.same_color_as(n2)
```

`Predicate` objects support logical conjunction and disjunction through the overloaded operators `&` and `|`. The following code shows how this may be used:

```
class Node:
    # ...
    @property
    def valid(self):
        color = self.color
        return ((color == 0) |
                (color == 1) |
                (color == 2))
```

The `valid` method of the `Node` class returns a predicate that is true when the node has one of three allowed colors. Notice that, apart from the unusual way of disjoining predicates, the `Node` class was written according to plain common sense object oriented programming practices. The only difference may be that, in the Coopy approach, the focus is on *property* rather than *behavior*.

Even if `Node` objects have a symbolic attribute, nothing prevents using them mostly like regular Python objects. Consider the following class `Edge`, for example:

```
from coopy import neg

class Edge:

    def __init__(self, a, b):
        self._a = a
        self._b = b

    @property
    def valid(self):
        same = self._a.same_color_as(self._b)
        return neg(same)
```

An edge is valid when the nodes it connects are not of the same color. Here, the `neg` function is used rather of the native `not` operator of Python. This is because the later cannot be overloaded, so a custom function had to be defined instead. The `neg` function of Coopy, when given a concrete boolean value, behaves just like the `not` operator, returning the complement of the input. When dealing with concrete values, therefore, `neg` and `not` can be used interchangeably. When given a `Predicate` object, however, `neg` returns a new one which is the logical negation of the original. When expecting a call to return a `Predicate`, then, `neg` should be used rather than `not`.

The key feature of `Predicate` objects is that they expose a method `require` which, when called, automatically compiles the predicate into a constraint and registers it with the solver back-end. To keep the Coopy programming model as natural and least intrusive as possible, the constraint solver is a global static entity implicitly accessed behind the curtains by the `require` method. This makes imposing constraints a relatively simple task, as shown in the next piece of code:

```
nodes = [Node() for i in range(N)]
edges = connect(nodes)

for node in nodes:
    node.valid.require()

for edge in edges:
    edge.valid.require()
```

The above code constructs a graph. A symbolic one, we may say, since its nodes still have `_color` attributes which are symbolic. These attributes are the unknowns of this graph coloring problem. Fortunately, Coopy makes finding (and using) a solution a very simple task.

2.3 Concretization

Having defined the object model and imposed the constraints, we can request Coopy to find a satisfying assignment of values for the symbolic attributes. This process in Coopy is called *concretization*. Concretization is achieved by calling the function `concretize`:

```
from coopy import concretize

concretize()
```

An important fact about concretization in Coopy is that it is completely transparent. When the `concretize` method is called, two things will happen: first, Coopy will engage the constraint solver to find values for the unknowns; second, Coopy will automatically and transparently update all symbolic variables present in the constraints with their computed value (they will become *concretized*). From then on, concretized symbolic variables will start behaving just like regular, concrete Python types. This means two things: *a*) retrieving the solution is just a matter of asking objects for their previously symbolic attributes (e.g. `node.color`), and *b*) the concretized network of objects is now *usable* as a concrete Python object. This casts Coopy not only as a relatively simple constraint modeling approach, but also as an interesting object instantiation mechanism for declarative object oriented programming, as will be shown in the upcoming section.

3 Application Examples

3.1 Declarative Object Factories

As mentioned in section 2.3, Coopy can be used as a declarative instantiation mechanism. Consider, for example, the code listing 1. Again, this code instantiates a k-colored graph. In this case, however, the graph construction procedure is encapsulated inside a function `construct_k_colored_graph` (which could be a method of some class, as well).

This functions receives three parameters: the maximum amount of colors allowed `k`, the amount of nodes `n`, and a probability `p` with which each pair of nodes will be connected. This algorithm then constructs a `k`-colored graph and returns it as a list of nodes (although it could be some `Graph` object as well).

Notice the very important fact that, while the connections between nodes are stated imperatively, the actual graph coloring algorithm is nowhere to be seen: this is transparently handled by Coopy and the constraint solver. After concretization, the graph is usable just like a plain Python object.

This example also shows how Coopy handles scopes. Since the constraint solver is a global static entity, some mechanism must be provided to avoid undesired clashes. When imported, Coopy is initialized with a default scope. A scope holds constraints and symbolic variable references, and is also invoked during the solving and concretization process. Aside from the default state, however, one may call the function `scope` to have Coopy instantiate a fresh new one with no variables or constraints

```python
import coopy
import random

class Node:

    def __init__(self):
        self._color = coopy.symbolic_int('c')
        self._neighbors = set()

    @property
    def color(self):
        return self._color

    @property
    def has_valid_connections(self):
        return coopy.all([self.color != n.color for n in self._neighbors])

    def add_direct_edge_towards(self, other):
        self._neighbors.add(other)

def construct_k_colored_graph(k, n, p):
    """
    Constructs a k colored graph of n nodes in which a pair
    of nodes shares an edge with probability 0 <= p <= 1.

    Note: This code is for demonstrative purposes only, as the solution
    for such a problem does not necessarily exist. In such cases,
    concretize will just throw an exception.
    """
    with coopy.scope():

        # Instantiate n nodes.
        nodes = [Node() for i in range(n)]

        # Connect nodes with probability p.
        for i in range(n-1):
            for j in range(i+1,n):
                a = nodes[i]
                b = nodes[j]
                if random.uniform(0,1) < p:
                    a.add_direct_edge_towards(b)
                    b.add_direct_edge_towards(a)

        # Impose restrictions over the nodes.
        for node in nodes:
            coopy.any([node.color == i for i in range(k)]).require()
            node.has_valid_connections.require()

        # Concretize the graph and return it as a list of nodes.
        coopy.concretize()
        return nodes
```

Code Listing 1: Declarative OOP with Coopy. The graph coloring algorithm is completely avoided.

whatsoever, and have it pushed into a stack. When creating variables, imposing constraints, or solving, Coopy always uses the scope at the top of the stack. Naturally, the scope is automatically popped out when the `with` block finalizes. This strategy allows factories to operate without worrying about their environment.

3.2 Bounded Model Checking

Coopy can also be used for bounded model checking [4]. A working strategy consists in defining a class that embodies a state transition with an `execute` method that imposes a disjunction of all valid outcomes. The following is an example of such a class:

```
class StateTransition:

    def __init__(self, bucket_a, bucket_b):
        self._a = bucket_a
        self._b = bucket_b

    def execute(self):
        a = self._a
        b = self._b
        outcome = False

        # Have both buckets advance
        # to a new state.
        a.advance()
        b.advance()

        # Fill one bucket completely.
        outcome |= a.filled & b.unchanged
        outcome |= a.unchanged & b.filled

        # Empty one bucket.
        outcome |= a.empty & b.unchanged
        outcome |= a.unchanged & b.empty

        # Pour the contents of one bucket
        # into the other.
        outcome |= a.poured_into(b)
        outcome |= b.poured_into(a)

        outcome.require()
```

This code, incidentally, is part of an example in Coopy's repository which solves the water jugs puzzle from the movie Die Hard 3. This example, while simple, shows how the mechanism can also be used to analyze the evolution of state machines in an object oriented fashion.

3.3 Type Synthesis

Coopy also takes advantage of the support provided by Z3 for declared sorts, quantifiers and uninterpreted functions. The key feature is that Coopy allows uninterpreted functions to be concretized as well. This can be used to synthesize types from axiomatic specifications, as shown in code examples 2 and 3. First, a `Boolean` class is defined with uninterpreted functions `+`, `*` and `~` represeting boolean operations. A `BooleanAlgebra` then defines elements `T` and `F`, and exposes an `axioms` method that returns a formula effectively specifying the axioms of the algebra. This formula is a predicate that may be required as a constraint. After concretization, then, Coopy will have assigned concrete interpretations to the boolean functions, allowing `Boolean` objects to be used like regular primitives. Having defined types `Boolean` and `BooleanAlgebra`, the remaining code would just be:

```
# Instantiate a boolean algebra.
algebra = BooleanAlgebra()

# Impose axioms.
algebra.axioms.require()

# Concretize. This resolves all boolean
# functions and gives identity to
# constants T and F.
coopy.concretize()

# Now the algebra can be used concretely.
T = algebra.T
F = algebra.F

assert(T != F and T.neg == F and T * F == F)
```

```
B = coopy.sort('B')

class Boolean(coopy.Evaluable):

    NOT = coopy.function('~', B, B)
    AND = coopy.function('*', B, B, B)
    OR  = coopy.function('+', B, B, B)

    def __init__(self, name=None, value=None):
        self._sym = value if value else coopy.symbolic(name if name else 'b', B)

    @property
    def value(self):
        return self._sym.value

    @property
    def neg(self):
        return Boolean.NOT(self, wrapper=Boolean)

    def __mul__(self, other):
        return Boolean.AND(self, other, wrapper=Boolean)

    def __add__(self, other):
        return Boolean.OR(self, other, wrapper=Boolean)

    def __eq__(self, other):
        return self._sym == other._sym

    def __ne__(self, other):
        return self._sym != other._sym
```

Code Listing 2: Type synthesis with Coopy: the `Boolean` type.

```python
from coopy import forall

class BooleanAlgebra:

    def __init__(self):
        # Instantiate the neutral elements of this boolean algebra.
        self.T = Boolean('T')
        self.F = Boolean('F')

    @property
    def axioms(self):

        axioms = True

        T = self.T
        F = self.F

        # Instantiate some constants to use as bound variables:
        p = Boolean('p')
        q = Boolean('q')
        r = Boolean('r')

        # Plain binary boolean algebra:
        axioms &= (T != F)
        axioms &= forall([p], (p == T) | (p == F))

        # Commutative property:
        axioms &= forall([p,q], p + q == q + p)
        axioms &= forall([p,q], p * q == q * p)

        # Distributive property:
        axioms &= forall([p,q,r], r + (p*q) == (r+p) * (r+q))
        axioms &= forall([p,q,r], r * (p+q) == (r*p) + (r*q))

        # Neutral elements:
        axioms &= forall([p], p + F == p)
        axioms &= forall([p], p * T == p)

        # Complements:
        axioms &= forall([p], p + p.neg == T)
        axioms &= forall([p], p * p.neg == F)

        return axioms
```

Code Listing 3: Type synthesis with Coopy: the `BooleanAlgebra` type.

4 Differences With Other CP Packages for Python

In the introduction it was stated that other constraint programming packages for Python are more procedural in nature, while Coopy is more object oriented. This section goes into detail into why that is the case, and explains what Coopy allows to do that other packages such as python-constraint [7] and Google OR-Tools [9] do not.

One of the key differences between Coopy and other packages is that, in Coopy, constraints have an entity of their own. This may not seem like much, but it has some important consequences: First, constraints become more powerful. They can override operators and encapsulate logic. Concretely, they can encapsulate the logic to *impose themselves*, ridding the programmer of the burden of passing a facade object around. Second, constraint objects can be returned even before being imposed. This allows to disperse the constraint building process cleanly across several methods.

In the Coopy programming model, constraints represent properties of objects, and compound objects combine the properties of their components. Overriding operators allows the act of casting the property of an object as a constraint to be equivalent to writing a method that returns whether said property is true or false. In addition, the fact that no facade object is required means that property methods can be written without any additional boilerplate: no extra parameters, no function calls, no transformation procedures, nor any other algorithmic overhead. Then, the act of conceiving and programming a constraint model becomes equivalent to the act of conceiving and programming an object model as is naturally done in the standard practice. This is an important goal of the system because it allows the programmer to apply his/her dexterity in building object models in full to construct constraint models. In other words, Coopy bridges the gap between object modeling and constraint modeling, and also mathematical modeling in general. This is also interesting because object modeling offers many important benefits such as modularity, reusability, scalability, and is also widely understood. This blend is not something that can be easily achieved with other packages without additional layers of abstraction. In essence, Coopy is not just a library: it also embodies an approach. In other words, Coopy not only provides a constraint solver interface: it also attempts to be a conceptual modeling tool.

There is also another important difference between Coopy and other packages: while other packages deem their job done once a solution has been provided, Coopy also focuses on how that solution will be used in the context of an object oriented program. The philosophy that guides this approach is the same that guides other characteristics of the package: the constraint model *is* the object model, no transformation is required. Therefore, Coopy injects the solution into the object model itself without any additional programming overhead. This allows for a clean implementation of the techniques discussed in section 3, namely declarative object factories and type synthesis. Other packages, on the other hand, require converting the output of the solver into an object model if such an usage is desired. Incidentally, they also require converting the object model *into* a constraint model if the system was first conceived as such.

5 Limitations

Aside from limitations proper of the Coopy programming model itself, the tool is unavoidably limited by the capabilities of the underly-

ing theorem prover as well. State explosion and unsatisfiable/unsolvable constraints are possible outcomes of a concretization attempt. In the later case, Coopy will just throw an exception.

The implementation of Coopy as a library also has a problem. First, the language does not allow overriding the logical operators `and`, `or` and `not`, which forces Coopy to depend on other operators and functions. The adoption of the operators `&` and `|` for conjunction and disjunction respectively, while reasonable, breaks the natural order of precedence and forces all connected constraints to be parenthesized e.g. as in `((x > y) & (x > 0)).require()`. On the other hand, if logical operators could be overriden, it *may* even be possible to encapsulate the fact that a certain method returns constraints, which could increase the reusability and transparency of the approach. This would require first dealing with the fact that logical operators also implement short-circuit evaluation, however, which may be desirable to turn off while compiling constraints. Such a deep integration would then require language and interpreter level support. In any case, such support would be an interesting thing to see in mainstream languages.

Another limitation in the approach may be the fact that the concretization process cannot just replace symbolic types by concrete types. After concretization, symbolic types start behaving as if they were concrete primitives, yet they are not. Therefore, symbolic types must be implemented to correctly emulate native primitives after concretization. The current version of Coopy does not support all possible operations that could be performed on primitive types. It also does not implement support for many types that the underlying solver can actually handle such as arrays and strings. Currently, Coopy supports only integers, booleans and reals, in addition to user declared sorts and uninterpreted functions.

Coopy also does not automatically *keep* a model valid after concretization. Further state changes can indeed lead to unmet constraints. While Coopy can be used to enforce valid state transitions, this process will typically not be time-efficient enough for general purpose programming.

Related to the previous point, constraint solving does carry a significant time overhead. Coopy, therefore, is probably not the best option for systems with strict performance requirements. In addition, while Coopy attempts to merge constraint modeling and object modeling seamlessly, it may well happen that the constraint model most naturally reflected by the object model is not also the easiest to solve.

Finally, Coopy cannot handle the synthesis of methods with side effects, as uninterpreted functions are essentially pure. While in the world of declarative programming this may be acceptable, in the world of imperative programming this will probably not be the case most of the times.

6 Conclusion

Despite its limitations, the Coopy approach proves to be an interesting way of writing software and solving mathematical problems. The tool is easy to use and easy to learn, and integrates smoothly into the familiar framework of object oriented programming. While declarative programming may not be as popular as imperative programming within the industry, introducing more declarative features in a way that blends in well with the object oriented programming style may lead to a more widespread adoption of these concepts.

7 Acknowledgements

Coopy was written as a tool to aid my research on software security in Fundación Dr. Manuel Sadosky, Argentina, as part of team STIC (Seguridad en Tecnologías de la Información y las Comunicaciones). I thank everyone in the team and the foundation for their support.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3), May 2018.

[3] N. Bjørner, L. de Moura, L. Nachmanson, and C. Wintersteiger. Programming Z3.

[4] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.

[5] C. Kreitz, K.-k. Lau, F. Intellektik, and T. Darmstadt. Logical Foundations for Declarative Object-oriented Programming. 07 1999.

[6] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, Jan. 2017.

[7] G. Niemeyer and S. Celles. python-constraint. Version 1.4.0. `https://pypi.org/project/python-constraint/`.

[8] F. Pérez and B. E. Granger. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[9] L. Perron and V. Furnon. OR-Tools, 2019. Google. Version 7.2. `https://developers.google.com/optimization/`.

[10] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, page 317–331, USA, 2010. IEEE Computer Society.

[11] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. 2016.

[12] F. van Harmelen, F. van Harmelen, V. Lifschitz, and B. Porter. *Handbook of Knowledge Representation*. Elsevier Science, San Diego, CA, USA, 2007.

[13] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.