Using Constraint Solvers as an oracle, with CPMpy

Prof. Tias Guns <tias.guns@kuleuven.be> @TiasGuns

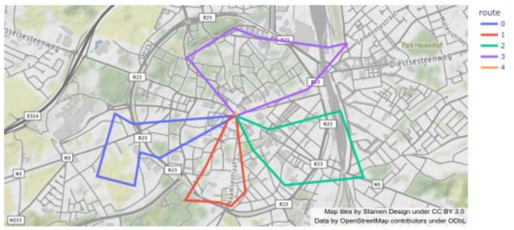
Emilio Gamba <emilio.gamba@vub.be>

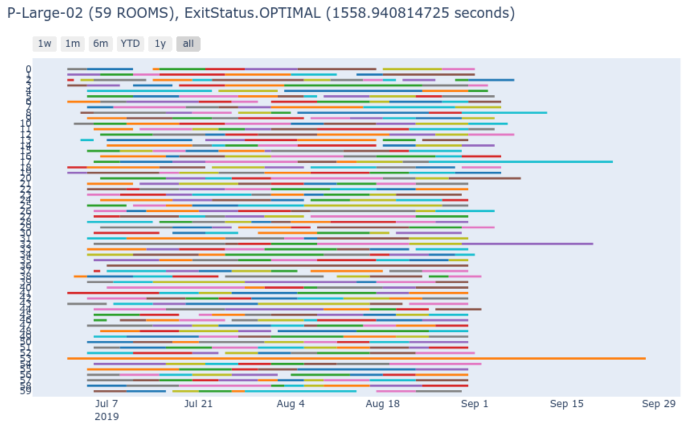
Ignace Bleukx <ignace.bleukx@kuleuven.be>

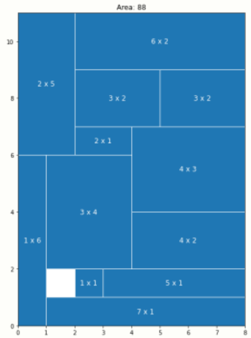
Constraint solving

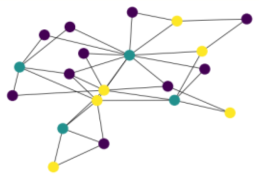
“Solving combinatorial optimisation problems”

- Vehicle Routing
- Scheduling
- Packing
- Other combinatorial problems







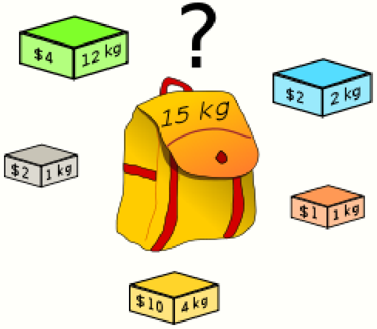


[Solved and visualized with the CPMpy constraint solving library]

Solving paradigm

Model + Solve

Modeling



Knapsack:

Model =

- Variables, with a domain
- Constraints over variables
- Optionally: an objective

- gr, bl, og, ye, gy :: {0,1}
- 12*gr + 2*bl + 1*og + 4*ye + 1*gy <= 15
- maximize(4*gr + 2*bl + 1*og + 10*ye + 2*gy)

Model.solve()

cpmpy.readthedocs.io/en/latest/

CPMpy
latest

Search docs

GETTING STARTED:

USER DOCUMENTATION:

API DOCUMENTATION:

CPMpy: Constraint Programming and Modeling in Python

CPMpy is a Constraint Programming and Modeling library in Python, based on numpy, with direct solver access.

Constraint Programming is a methodology for solving combinatorial optimisation problems like assignment problems or covering, packing and scheduling problems. Problems that require searching over discrete decision variables.

CPMpy allows to model search problems in a high-level manner, by defining decision variables and constraints and an objective over them (similar to MiniZinc and Essence'). You can freely use numpy functions and indexing while doing so. This model is then automatically translated to state-of-the-art solver like or-tools, which then compute the optimal answer.

Source code and bug reports at https://github.com/CPMpy/cpmpy

Getting started:

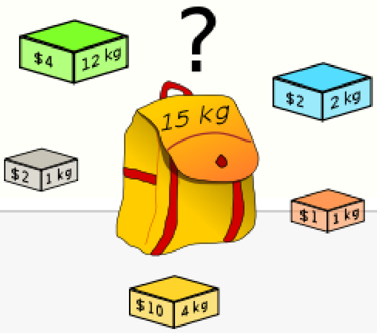
User Documentation:

API documentation:

https://cpmpy.readthedocs.io/

Modeling

Knapsack:



Model =

- Variables, with a domain
- Constraints over variables
- Optionally: an objective

Model.solve()

```
model = Model()

gr,bl,og,ye,gy = boolvar(shape=5)

model += (12*gr + 2*bl + 1*og + 4*ye + 1*gy <= 15)

model.maximize(4*gr + 2*bl + 1*og + 10*ye + 2*gy)


model.solve()
```

```
print(gr.value(), bl.value(), og.value(), ye.value(), gy.value())
```

```
0 1 1 1 1
```

Modeling

Also satisfaction problems, e.g. *sudoku*


```
e = 0 # value for empty cells
given = np.array([
    [e, e, 2,  4, 1, e,  e, e, 5],
    [1, e, 4,  3, e, e,  e, e, e],
    [e, 8, e,  2, 7, 5,  3, 4, 1],

    [e, e, e,  e, 3, 1,  e, e, e],
    [7, 9, e,  e, e, e,  e, 2, e],
    [e, e, e,  e, e, e,  e, e, e],

    [e, e, e,  e, e, 4,  e, 6, e],
    [5, e, e,  8, e, e,  4, e, 9],
    [e, 4, e,  1, e, 3,  5, 7, e]])
```

```
model = Model()

# Variables
puzzle = intvar(1, 9, shape=given.shape, name="puzzle")

# Constraints on rows and columns
model += [AllDifferent(row) for row in puzzle]
model += [AllDifferent(col) for col in puzzle.T]

# Constraints on blocks
for i in range(0,9, 3):
    for j in range(0,9, 3):
        model += AllDifferent(puzzle[i:i+3, j:j+3])

# Constraints on values (cells that are not empty)
model += (puzzle[given!=e] == given[given!=e])


model.solve()
```

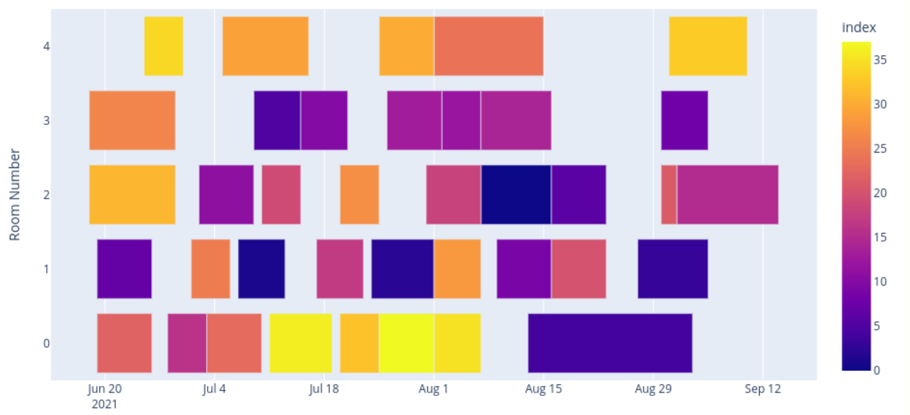
Other examples: room scheduling

Demo

https://github.com/CPMpy/cpmpy/blob/master/examples/room_assignment.ipynb

Example: room scheduling (backup slide)

```python
def model_rooms(df, max_rooms, verbose=True):
    n_requests = len(df)

    # All requests must be assigned to one out of the rooms (same room during entire period).
    requestvars = intvar(0, max_rooms-1, shape=(n_requests,))

    m = Model()

    # Some requests already have a room pre-assigned
    for idx, row in df.iterrows():
        if not pd.isna(row['room']):
            m += (requestvars[idx] == int(row['room']))

    # A room can only serve one request at a time.
    # <=> requests on the same day must be in different rooms
    for day in pd.date_range(min(df['start']), max(df['end'])):
        overlapping = df[(df['start'] <= day) & (day < df['end'])]
        if len(overlapping) > 1:
            m += AllDifferent(requestvars[overlapping.index])

    return (m, requestvars)
```



Solving

model.solve()

Depends on solver family...

- SAT: *Boolean* decision variables; *clauses* as constraints
- MIP: *Integer* decision variables; *linear* constraints
- CP: *Integer* decision variables; *logical, mathematical, global* constraints

The changing role of solvers

Holy Grail: user specifies, solver solves [Freuder,1997]

I think we reached it… MiniZinc, Essence'

"Beyond NP" → Constraint Solver as an **oracle**

- Use CP solver to solve subproblem of larger algorithm
- Iteratively build-up and solve a problem until failure
- Integrate neural network predictions (structured output prediction)
- Generate proofs, explanations, or counterfactual examples, …

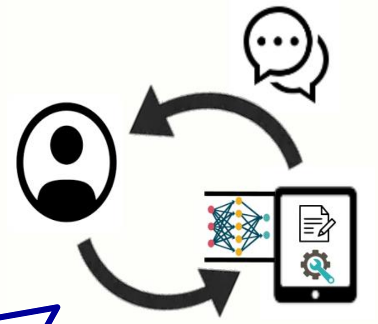
Solving paradigm, taking the human in the loop

Model + Solve


Opt. expert


Domain experts

CHAT-Opt:
Conversational **H**uman-**A**ware **T**echnology for **Opt**imisation



Towards **co-creation** of constrained optimisation solutions

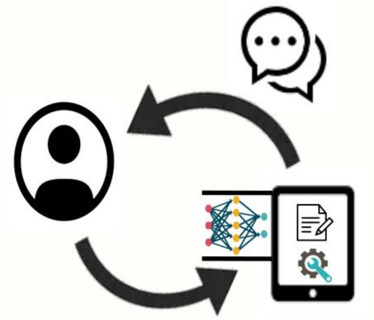
- Solver that learns from user and environment
- Towards conversational: explanations and stateful interaction

https://people.cs.kuleuven.be/~tias.gun

@TiasGuns

Hiring post-docs!

Towards conversational solving

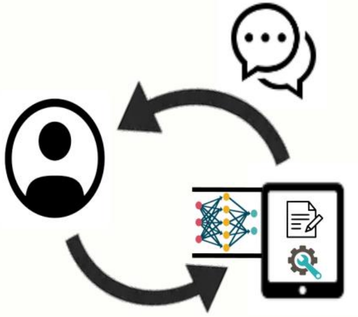


Asking for explanations

- Why is there no solution?
- How is this solution obtained?
- Why is X part of the solution?
- What are possible alternatives?
- What if Y should be part of the solution?

=> requires "Constraint Solving as oracle"

Conversational Human-Aware Technology for Optimisation



What would the ideal Constraint Solving system be?

- Efficient repeated solving
  => Incremental
- Use CP/SAT/MIP or any combination
  => solver independent and multi-solver
- Easy integration with Machine Learning libraries
  => Python and numpy arrays

What would the ideal Constraint Solving system be?

- **Efficient repeated solving**
  => Incremental

- Use CP/SAT/MIP or any combination
  => solver independent and multi-solver

- Easy integration with Machine Learning libraries
  => Python and numpy arrays

Incrementality

Solving:

- MIP: can add constraints, change objective
  (mechanisms not documented, e.g. start from previous basis)
- SAT: *assumption* variables: can be toggled on/off when calling solve, adding constraints
  (reuses learned clauses, variable activity)
- CP: if CP-SAT, assumption variables like SAT, adding constraints and changing objective
- SMT: all of the above and push/pop of constraints (Z3)

Modeling?

- Only if using solver API directly...
- With CPMpy: part of the high-level modeling language!

Multiple solutions

Returns True (sol. found) or False (no solution)

```
x = intvar(0,3, shape=2)
m = Model(x[0] > x[1])

while m.solve():
    print(x.value())
    m += ~all(x == x.value()) # block solution
```

```
[3 0]
[3 1]
[3 2]
[2 0]
[1 0]
[2 1]
```

Adds constraint to model (even if already solved before)

Alternative (diverse) solutions

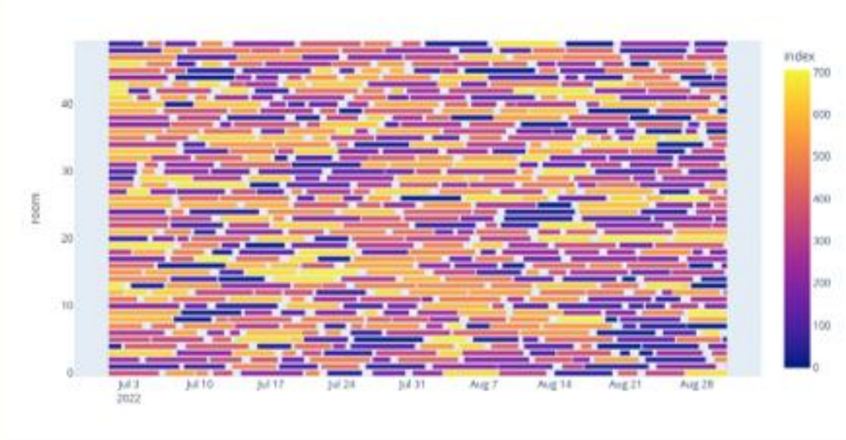
```python
# a diversity measure, hamming distance
def hamm(x, y):
    return sum(x != y)

x = intvar(0,3, shape=2)
m = Model(x[0] > x[1])

store = []
while m.solve():
    print(len(store), ":", x.value())
    m += ~all(x == x.value()) # block solution
    store.append(x.value())
    # maximize number of elements that are different
    m.maximize(sum(hamm(x, sol) for sol in store))
```

```
0 : [3 0]
1 : [2 1]
2 : [1 0]
3 : [3 2]
4 : [2 0]
5 : [3 1]
```

Can change obj. function (even if already solved before)

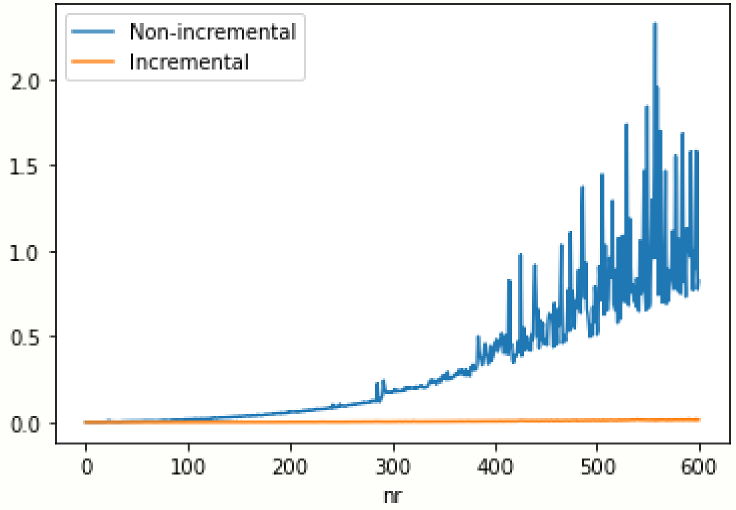
Incremental room assignment problem

```python
def model_rooms(df, max_rooms, verbose=True):
    n_requests = len(df)

    # All requests must be assigned to one out of the rooms (same room during entire period).
    requestvars = intvar(0, max_rooms-1, shape=(n_requests,))

    m = Model()

    # Some requests already have a room pre-assigned
    for idx, row in df.iterrows():
        if not pd.isna(row['room']):
            m += (requestvars[idx] == int(row['room']))

    # A room can only serve one request at a time.
    # <=> requests on the same day must be in different rooms
    for day in pd.date_range(min(df['start']), max(df['end'])):
        overlapping = df[(df['start'] <= day) & (day < df['end'])]
        if len(overlapping) > 1:
            m += AllDifferent(requestvars[overlapping.index])

    return (m, requestvars)
```



Assume requests come in sequentially.

Compute solution on every new request.



What would the ideal Constraint Solving system be?

- Efficient repeated solving

  => Incremental

- **Use CP/SAT/MIP or any combination**

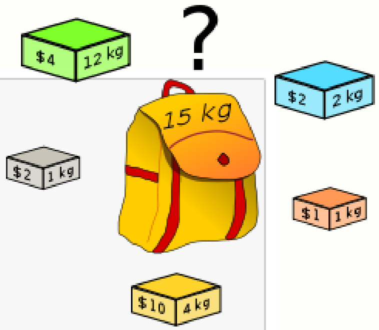
  => solver independent and multi-solver

- Easy integration with Machine Learning libraries

  => Python and numpy arrays

Multi-solver

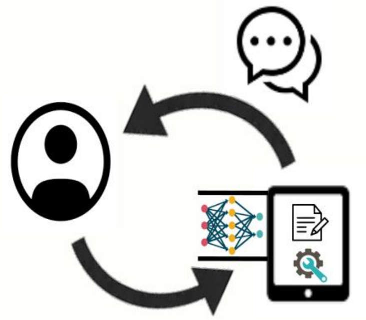
Same syntax, plus can reuse variables and their values




```
m_ort = SolverLookup.get("ortools", model_knapsack)
m_ort.solve()
print("\nOrtools:", m_ort.status(), ":", m_ort.objective_value(), items.value())

m_grb = SolverLookup.get("gurobi", model_knapsack)
m_grb.solve()
print("\nGurobi:", m_grb.status(), ":", m_grb.objective_value(), items.value())

# use ortools to verify the gurobi solution
m_ort += (items == items.value())
print("\tGurobi's is a valid solution according to ortools:", m_ort.solve())
```

```
Ortools: ExitStatus.OPTIMAL (0.001146096 seconds) : 32.0 [ True False False  True  True  True  True  True]

Gurobi: ExitStatus.OPTIMAL (0.0003108978271484375 seconds) : 32.0 [ True False  True False  True  True  True  Tru
e]
	Gurobi's is a valid solution according to ortools: True
```

Conversational **H**uman-**A**ware **T**echnology for **Opt**imisation



What would the ideal CP system be?

- Efficient repeated solving

  => Incremental

- Use CP/SAT/MIP or any combination

  => solver independent and multi-solver

- **Easy integration with Machine Learning libraries**

  => Python and numpy arrays

  Not covered, but see

  https://github.com/CPMpy/cpmpy/blob/master/examples/advanced/visual_sudoku.ipynb

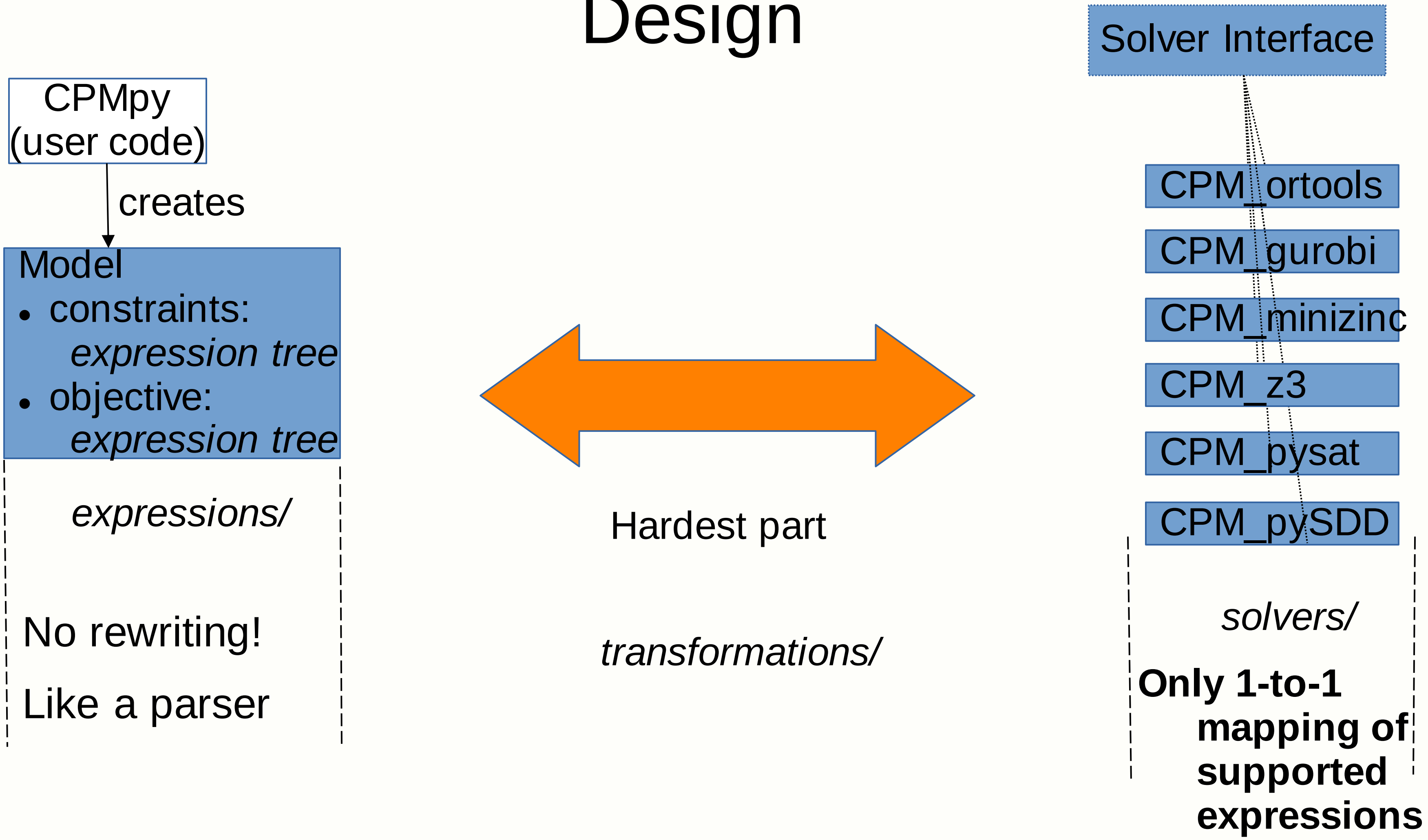
3 short slides on CPMpy's design

Design principle:

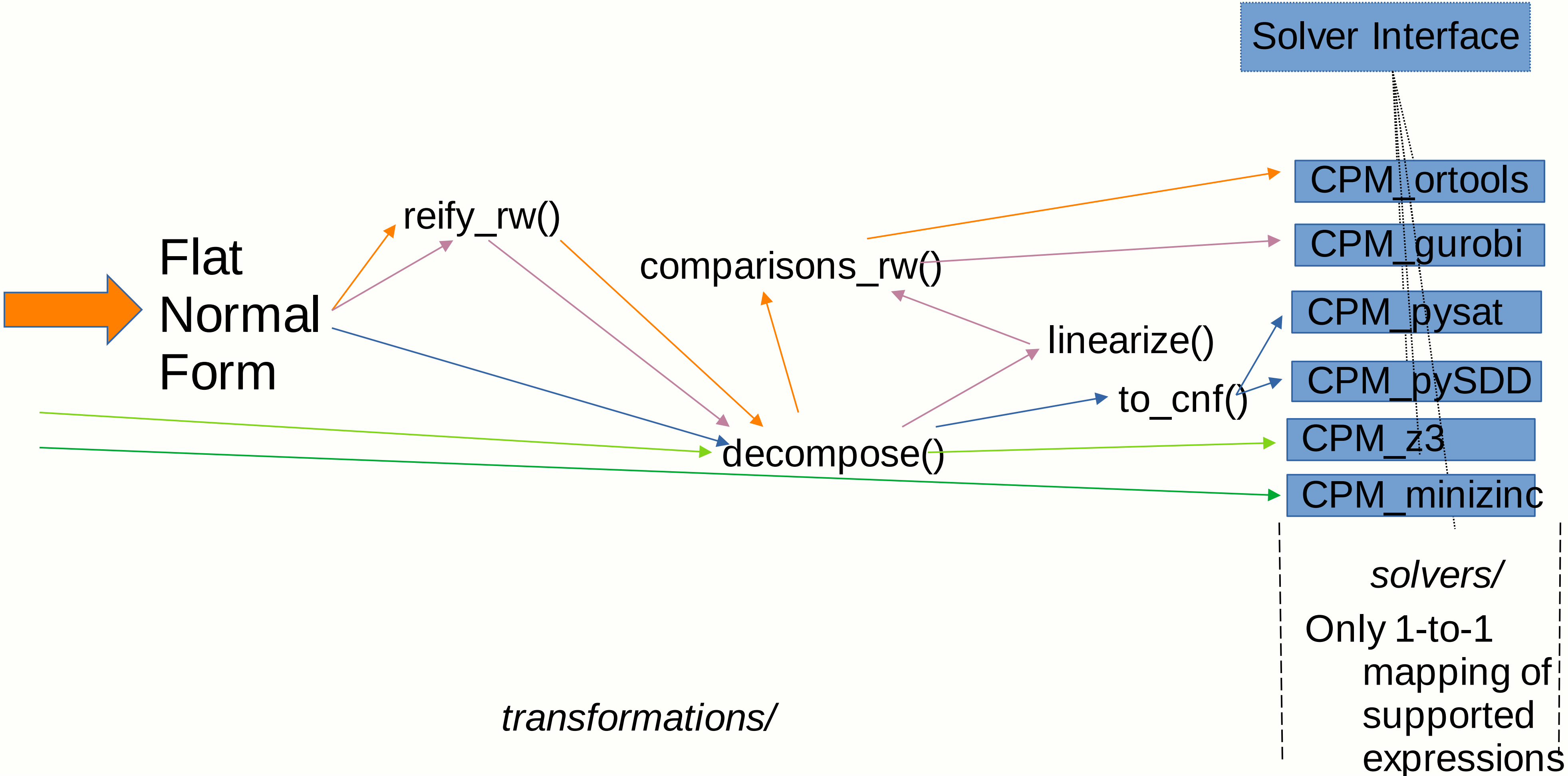
Aim to be a thin layer on top of solver API

Central concept: CPMpy expression

Design


CPMpy
(user code)
creates
Model
• constraints: *expression tree*
• objective: *expression tree*
expressions/
No rewriting!
Like a parser
Hardest part
transformations/
Solver Interface
CPM_ortools
CPM_gurobi
CPM_minizinc
CPM_z3
CPM_pysat
CPM_pySDD
solvers/
Only 1-to-1 mapping of supported expressions

Transformations in a nutshell



Solvers

CPMpy only interfaces to Python APIs

Key principle: solver can implement any subset of expressions!

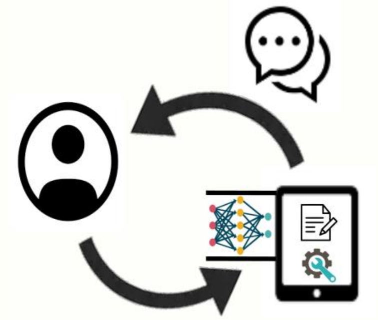
Solvers can also choose to:

- Support assumptions or not
- Be incremental or not
- Expose own solver parameters

Currently:
- ortools
- pysat
- minizinc
- gurobi
- pySDD

Near future: ExactSolver, Z3
Wishlist: Mistral2, Geas, Gecode

Towards conversational solving



Asking for <u>explanations</u>

- Why is there no solution?
- How is this solution obtained?
- Why is X part of the solution?
- What are possible alternatives?
- What if Y should be part of the solution?

=> requires "Constraint Solving as oracle"