Leveraging Block-Composable Optimization Modeling Environments for Transmission Switching and Unit Commitment

John D. Siirola,[1] Jean-Paul Watson,[1] and David L. Woodruff [2]

[1] Discrete Math & Complex Systems Department
Sandia National Laboratories
Albuquerque, NM USA

[2] Graduate School of Management
University of California, Davis
Davis, CA USA

Increasing Real-Time and Day-Ahead Market Efficiency through Improved Software
25 June 2012

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This is a talk on *modeling environments*

- Our premise:
  - Optimization (math programming; MP) is critical for grid planning and operations
  - Typical models are tough to create and tougher to understand
  - We increasingly leverage *problem-specific structure* to solve harder (bigger, more complex) problems effectively
- Our approach:
  - New MP modeling environment (Pyomo)
    - *Extensible*: new modeling constructs
    - *Powerful*: develop new native solvers, heuristic methods
    - *Open*: (1) transparent;
      solvers, heuristics, etc. can interrogate and manipulate model
    - *Open*: (2) freely distributable;
      researchers, vendors, operators can share models

The Challenge: MP is dense and subtle

$$\text{Minimize}: \quad \sum_t \sum_g \left(c_g P_{g0t} + c_g^{SU} v_{gt} + c_g^{SD} w_{gt} \right) \tag{1}$$

$$\text{S.t.} \quad \theta^{\min} \le \theta_{nct} \le \theta^{\max}, \quad \forall\, n, c, t \tag{2}$$

$$\sum_{\forall\, k(n,.)} P_{kct} - \sum_{\forall\, k(.,n)} P_{kct} + \sum_{\forall\, g(n)} P_{g0t} = d_{nt},$$
$$\forall\, n, \quad c = 0, \text{ transmission contingency states } c, t \tag{3a}$$

$$\sum_{\forall\, k(n,.)} P_{kct} - \sum_{\forall\, k(.,n)} P_{kct} + \sum_{\forall\, g(n)} P_{gct} = d_{nt},$$
$$\forall\, n, \text{ generator contingency states } c, t \tag{3b}$$

$$P_{kc}^{\min} N1_{kc} z_{kt} \le P_{kct} \le P_{kc}^{\max} N1_{kc} z_{kt}, \quad \forall\, k, c, t \tag{4}$$

$$B_k(\theta_{nct} - \theta_{mct}) - P_{kct} + (2 - z_{kt} - N1_{kc}) M_k \ge 0, \quad \forall\, k, c, t \tag{5a}$$

$$B_k(\theta_{nct} - \theta_{mct}) - P_{kct} - (2 - z_{kt} - N1_{kc}) M_k \le 0, \quad \forall\, k, c, t \tag{5b}$$

$$P_g^{\min} N1_{gc} u_{gt} \le P_{gct} \le P_g^{\max} N1_{gc} u_{gt}, \quad \forall\, g, c, t \tag{6}$$

$$v_{g,t} - w_{g,t} = u_{g,t} - u_{g,t-1}, \quad \forall\, g, t \tag{7}$$

$$\sum_{q=t-UT_g+1}^{t} v_{g,q} \le u_{g,t}, \quad \forall\, g, t \in \{UT_g, \ldots, T\} \tag{8}$$

$$\sum_{q=t-DT_g+1}^{t} w_{g,q} \le 1 - u_{g,t}, \forall\, g, t \in \{DT_g, \ldots, T\} \tag{9}$$

$$P_{g0t} - P_{g0,t-1} \le R_g^{+} u_{g,t-1} + R_g^{SU} v_{g,t}, \quad \forall\, g, t \tag{10}$$

$$P_{g0,t-1} - P_{g0,t} \le R_g^{-} u_{g,t} + R_g^{SD} w_{g,t}, \quad \forall\, g, t \tag{11}$$

$$P_{gct} - P_{g0,t} \le R_g^{+}, \quad \forall\, g, c, t \tag{12}$$

$$P_{g0,t} N1_{gc} - P_{gct} \le R_g^{-}, \quad \forall\, g, c, t \tag{13}$$

$$0 \le v_{g,t} \le 1, \quad \forall\, g, t \tag{14}$$

$$0 \le w_{g,t} \le 1, \quad \forall\, g, t \tag{15}$$

$$u_{g,t} \in \{0, 1\}, \quad \forall\, g, t \tag{16}$$

Hedman, et al., "Co-Optimization of Generation Unit Commitment and Transmission Switching With N-1 Reliability," *IEEE Trans Power Systems*, 25(2), pp.1052-1063, 2010

The Challenge: MP is dense and subtle

$$\text{Minimize}: \sum_t \sum_g \left(c_g P_{g0t} + c_g^{SU} v_{gt} + c_g^{SD} w_{gt}\right) \tag{1}$$

$$\text{S.t.} \quad \theta^{\min} \le \theta_{nct} \le \theta^{\max}, \quad \forall\, n, c, t$$

$$\sum_{\forall\, k(n,.)} P_{kct} - \sum_{\forall\, k(.,n)} P_{kct} + \sum_{\forall\, g(n)} P_{g0t} = d_{nt},$$
$$\forall\, n, \quad c = 0, \text{ transmission contingency states } c, t$$

$$\sum_{\forall\, k(n,.)} P_{kct} - \sum_{\forall\, k(.,n)} P_{kct} + \sum_{\forall\, g(n)} P_{gct} = d_{nt},$$
$$\forall\, n, \text{ generator contingency states } c, t$$

$$P_{kc}^{\min} N1_{kc} z_{kt} \le P_{kct} \le P_{kc}^{\max} N1_{kc} z_{kt}, \quad \forall\, k, c, t$$

$$B_k(\theta_{nct} - \theta_{mct}) - P_{kct} + (2 - z_{kt} - N1_{kc}) M_k \ge 0, \quad \forall\, k, c, t$$

$$B_k(\theta_{nct} - \theta_{mct}) - P_{kct} - (2 - z_{kt} - N1_{kc}) M_k \le 0, \quad \forall\, k, c, t \tag{5b}$$

$$P_g^{\min} N1_{gc} u_{gt} \le P_{gct} \le P_g^{\max} N1_{gc} u_{gt}, \quad \forall\, g, c, t \tag{6}$$

$$v_{g,t} - w_{g,t} = u_{g,t} - u_{g,t-1}, \quad \forall\, g, t \tag{7}$$

$$\sum_{q=t-UT_g+1}^{t} v_{g,q} \le u_{g,t}, \quad \forall\, g, t \in \{UT_g, \ldots, T\} \tag{8}$$

$$\sum_{q=t-DT_g+1}^{t} w_{g,q} \le 1 - u_{g,t}, \forall\, g, t \in \{DT_g, \ldots, T\} \tag{9}$$

$$P_{g0t} - P_{g0,t-1} \le R_g^+ u_{g,t-1} + R_g^{SU} v_{g,t}, \quad \forall\, g, t \tag{10}$$

$$P_{g0,t-1} - P_{g0,t} \le R_g^- u_{g,t} + R_g^{SD} w_{g,t}, \quad \forall\, g, t \tag{11}$$

$$P_{gct} - P_{g0,t} \le R_g^+, \quad \forall\, g, c, t \tag{12}$$

$$P_{g0,t} N1_{gc} - P_{gct} \le R_g^-, \quad \forall\, g, c, t \tag{13}$$

$$0 \le v_{g,t} \le 1, \quad \forall\, g, t \tag{14}$$

$$0 \le w_{g,t} \le 1, \quad \forall\, g, t \tag{15}$$

$$u_{g,t} \in \{0, 1\}, \quad \forall\, g, t \tag{16}$$

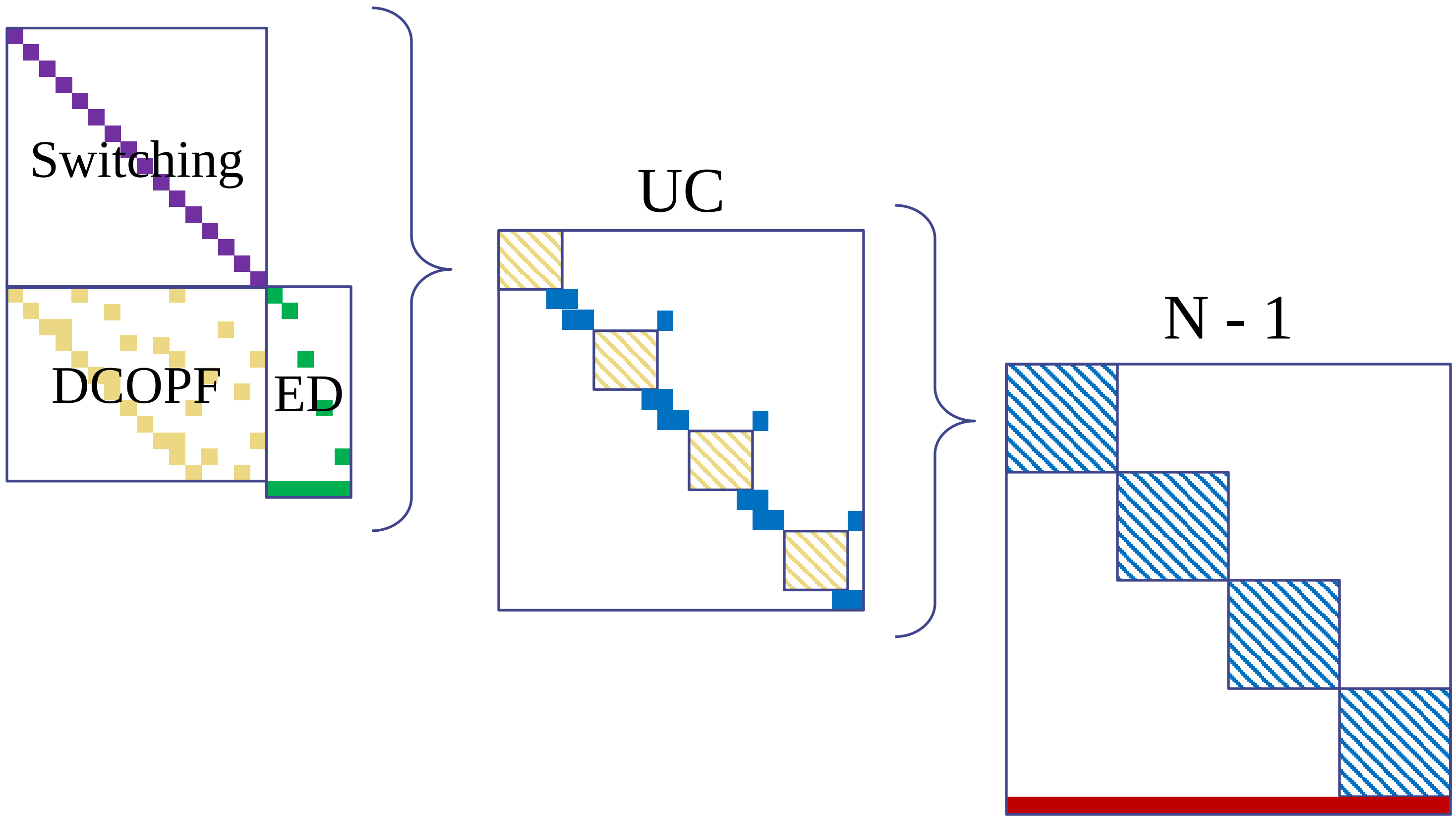
To a first approximation:

- DCOPF
- Economic dispatch
- Unit commitment
- Transmission switching
- N-1 contingency

Hedman, et al., "Co-Optimization of Generation Unit Commitment and Transmission Switching With N-1 Reliability," *IEEE Trans Power Systems*, 25(2), pp.1052-1063, 2010

(Nonobvious) Inherent structure

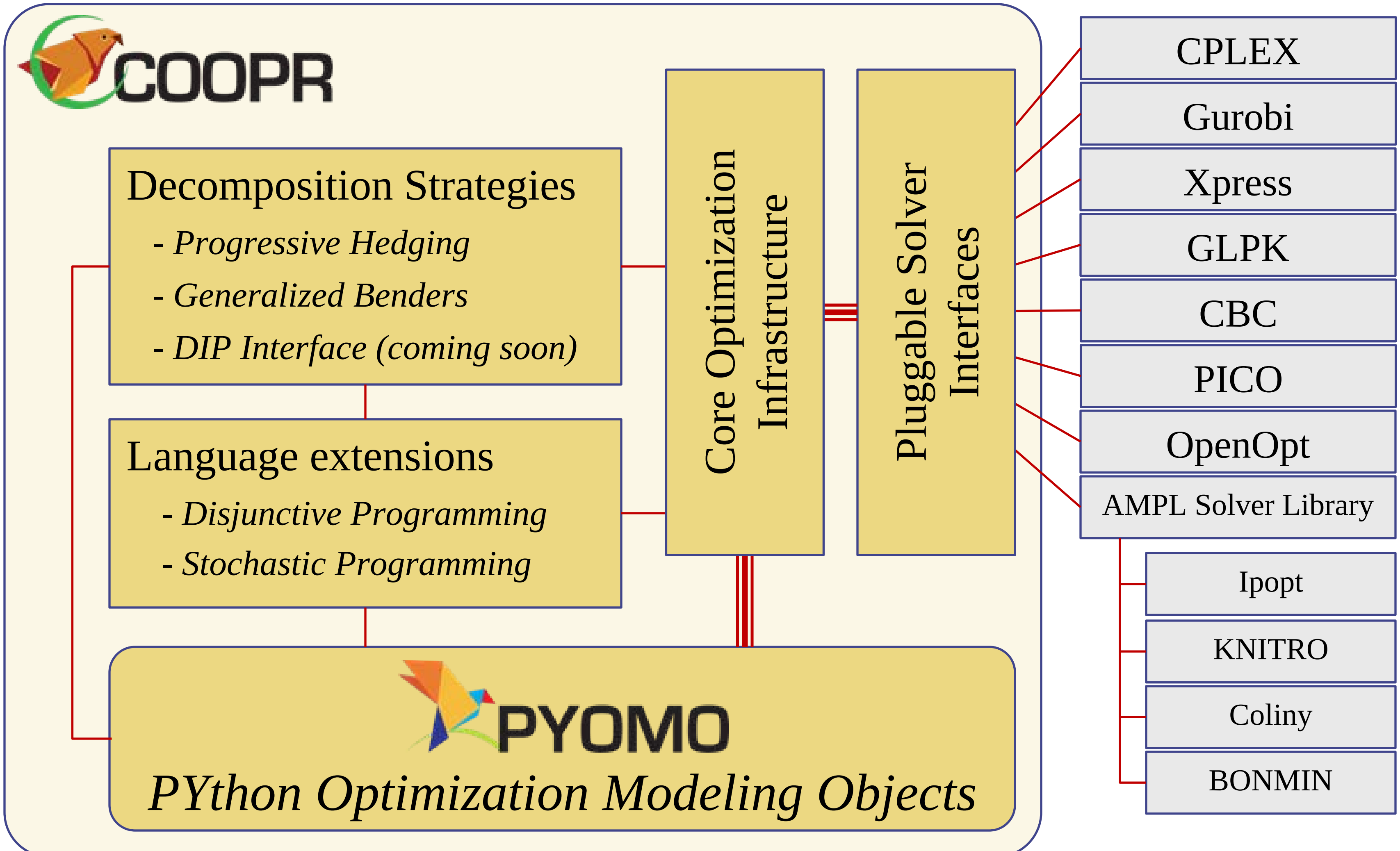
- Layered (nested) model complexity



Block-oriented modeling

- "Blocks"
  - Collections of model components
    - Var, Param, Set, Constraint, etc.
  - Blocks may be arbitrarily nested
- Why blocks?
  - Support reusable modeling components
  - Express distinctly modeled concepts as distinct objects
  - Manipulate modeled components as distinct entities
  - Explicitly expose model structure (e.g., for decomposition)
- Prior art
  - Ubiquitous in the simulation community
  - Rare in Math Programming environments
    - *Notable exceptions:* ASCEND, JModelica.org

Coopr: *a COmmon Optimization Python Repository*



Pyomo overview

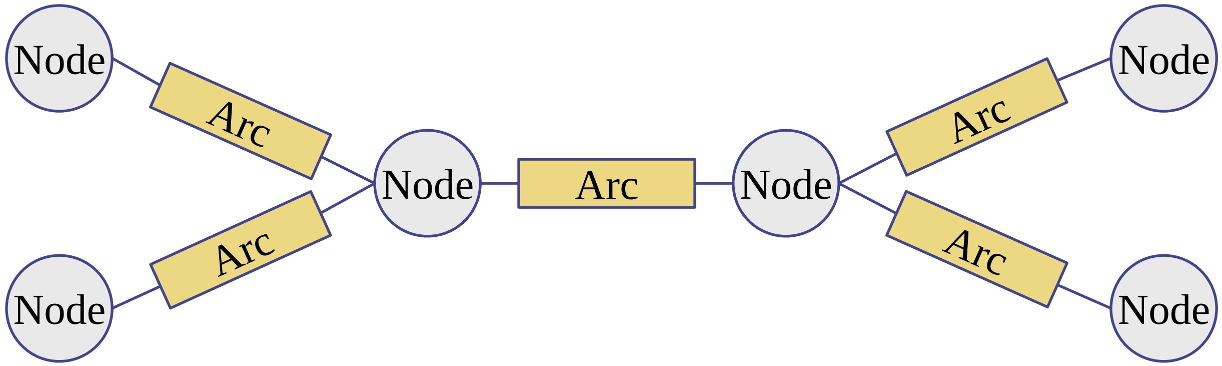
- Formulating optimization models natively within Python
  - Provides a natural syntax to describe mathematical models
  - Can formulate large models with a concise syntax
  - Separates modeling and data declarations
  - Enables data import and export in commonly used formats

- Highlights:
  - Clean syntax
  - Python scripts provide a flexible context for exploring the structure of Pyomo models
  - Leverage high-quality third-party Python libraries, e.g., SciPy, NumPy, MatPlotLib

```
from coopr.pyomo import *

m = ConcreteModel()

m.x1 = Var()
m.x2 = Var(bounds=(-1,1))
m.x3 = Var(bounds=(1,2))

m.obj = Objective(
   sense = minimize,
   expr = m.x1**2 + (m.x2*m.x3)**4 +
            m.x1*m.x3 + m.x2 +
            m.x2*sin(m.x1+m.x3) )
model = m
```

Structured modeling with blocks

- Capture connected block structure, e.g., *network flow*



- Block interface through *connectors* (group of variables)
- Block implementation independent of network definition

Domain	Node	Arc	Connector Vars
Fluid flow	Mass balance	Pressure Drop	Pressure; Volumetric flow
AC Power flow	KCL	Active power transfer; Reactive power transfer	Phase angle; Active power flow; Reactive power flow

DC OPF: transmission (line) model

```
def dc_line_rule(line, id):
    line.B            = Param()
    line.Limit        = Param()

    line.V_angle_in  = Var()
    line.V_angle_out = Var()
    line.Power        = Var( bounds= ( -line.Limit, line.Limit ) )

    line.power_flow = Constraint( expr= \
        line.Power == line.B*(line.V_angle_in - line.V_angle_out) )

    line.IN = Connector( initialize= \
        { “Power”: -line.Power, “V_angle”: line.V_angle_in } )

    line.OUT = Connector( initialize= \
        { “Power”: line.Power, “V_angle”: line.V_angle_out } )
```

General power flow model (1 / 2)

```
from coopr.pyomo import *

model = AbstractModel()

model.BUSES = Set()
model.LINES = Set()
model.GENERATORS = Set()
model.ENDPOINTS = Set(initialize=[“IN”, “OUT”])

model.links = Param( model.LINES, model.ENDPOINTS )
```


Sandia National Laboratories

General power flow model (2 / 2)

```
from power_flow import \
        dc_line_rule as line_rule, \
        dc_bus_rule as bus_rule, \
        dc_generator_rule as generator_rule
```

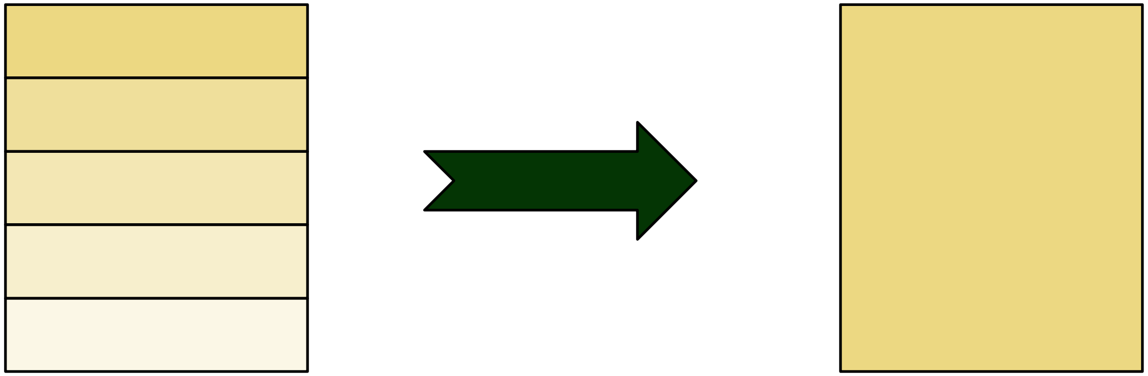
Only domain-specific component
(Note: we have only shown the line rule and not the bus or generator rules)

```
model.bus       = Block( model.BUSES, rule=bus_rule )
model.line      = Block( model.LINES, rule=line_rule )
model.generator = Block( model.GENERATORS, rule=generator_rule )

def _network(model, l, end):
   if endpoint == 'IN':
      return model.line[l].IN == model.bus[ value(model.links[l, end]) ].PORT
   else:
      return model.line[l].OUT == model.bus[ value(model.links[l, end]) ].PORT
model.network = Constraint(model.LINES, model.ENDPOINTS, rule=_network)

def _generator_placement(model, g):
   return model.generator[g].OUT == model.bus[ value(model.generator[g].bus) ].PORT
model.generator_placement = Constraint(model.GENERATORS, rule=_generator_placement)
```


Sandia National Laboratories

Solving block models

1) Construct hierarchical model
   - Generate blocks (Variables + Internal constraints)
   - "Connect" blocks by forming constraints over block connectors
2) Use a *model transformation* to "flatten" the model
   - Replicates connector constraints for each variable in connector
   - Generates aggregating constraints
   - (Eliminates redundant variables)



Sandia National Laboratories

Model libraries: switching to ACOPF is trivial

```
from power_flow import  ac_line_rule as line_rule, \
                        ac_bus_rule as bus_rule, \
                        ac_generator_rule as generator_rule
```

```
model.bus       = Block( model.BUSES, rule=bus_rule )
model.line      = Block( model.LINES, rule=line_rule )
model.generator = Block( model.GENERATORS, rule=generator_rule )

def _network(model, l, end):
   if endpoint == ‘IN’:
      return model.line[l].IN == model.bus[ value(model.links[l, end]) ].PORT
   else:
      return model.line[l].OUT == model.bus[ value(model.links[l, end]) ].PORT
model.network = Constraint(model.LINES, model.ENDPOINTS, rule=_network)

def _generator_placement(model, g):
   return model.generator[g].OUT == model.bus[ value(model.generator[g].bus) ].PORT
model.generator_placement = Constraint(model.GENERATORS, rule=_generator_placement)
```

Manipulating model blocks

- Generalized Disjunctive Programming (GDP)
  - Switching entire blocks on/off through binary variables

- Introduce new modeling components:
  - “Disjunct”
    - a new form of model block
  - “Disjunction”
    - a new constraint for enforcing logical XOR over disjunctive sets

$$
\begin{aligned}
\min \quad & \sum\nolimits_k c_k + f(x) \\
s.t. \quad & g(x) \le 0 \\
& \bigvee_{i \in D_k} \left[\begin{array}{c} Y_{ik} \\ h_{ik}(x) \le o \\ c_k = \gamma_{ik} \end{array} \right] \\
& \Omega(Y) = true \\
& Y_{ik} \in \{true, false\}
\end{aligned}
$$

Creating an “open line” model

```
def open_dc_line_rule(line):
    line.V_angle_in  = Var()
    line.V_angle_out = Var()
    line.Power       = Var()

    line.power_flow = Constraint( expr= line.Power == 0 )

    line.IN = Connector( initialize= \
        { “Power”: -line.Power, “V_angle”: line.V_angle_in } )

    line.OUT = Connector( initialize= \
        { “Power”: line.Power, “V_angle”: line.V_angle_out } )
```

Creating a “switchable line”

```
def switchable_dc_line_rule(line, id):
    line.V_angle_in  = Var()
    line.V_angle_out = Var()
    line.Power_in    = Var()
    line.Power_out   = Var()

    line.IN = Connector( initialize= \
        { “Power”: line.Power_in, “V_angle”: line.V_angle_in } )
    line.OUT = Connector( initialize= \
        { “Power”: line.Power_out, “V_angle”: line.V_angle_out } )

    line.Closed = Disjunct( rule=dc_line_rule )
    line.Open   = Disjunct ( rule=open_dc_line_rule )
    line.Switch = Disjunction( initialize=[line.Closed, line.Open] )

    line.connections = ConstraintList()
    for block in ( line.Open, line.Closed ):
        line.connections.add( line.IN = block.IN )
        line.connections.add( line.OUT = block.OUT )
```

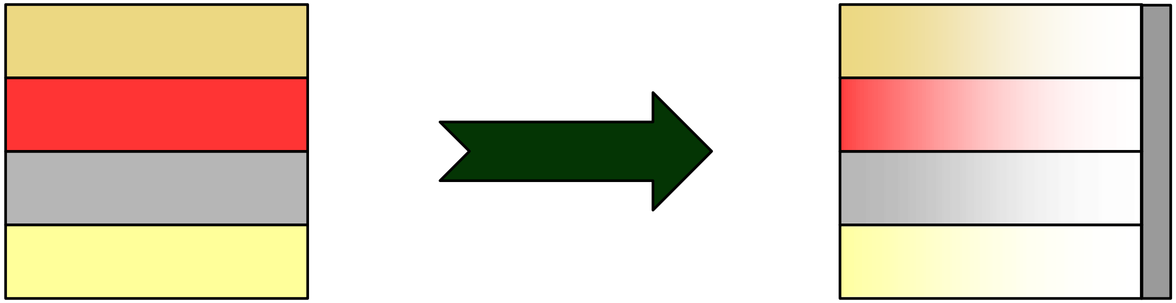
Creating a transmission switching model

```
from power_flow import  switchable_dc_line_rule as line_rule, \
                        dc_bus_rule as bus_rule, \
                        dc_generator_rule as generator_rule
```

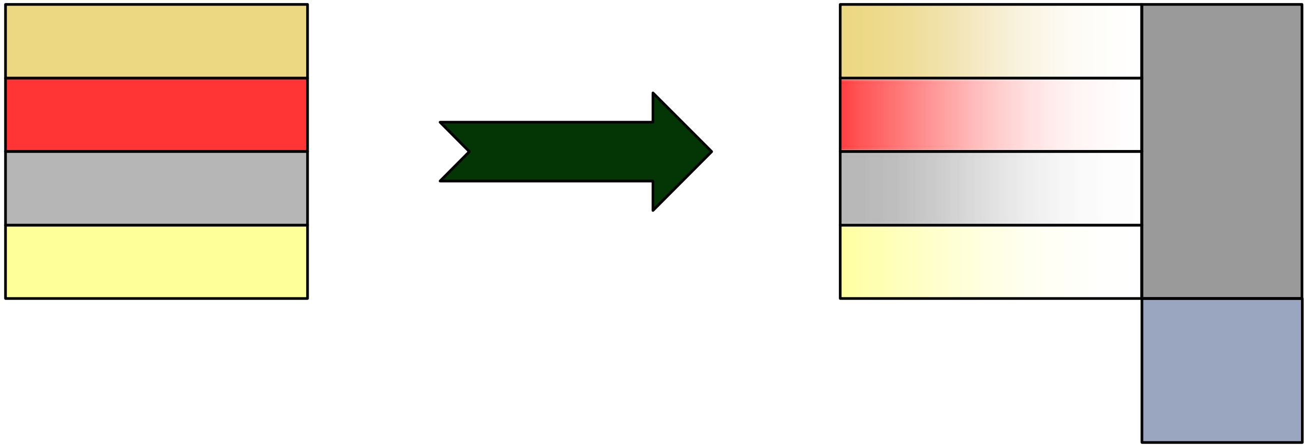
```
model.bus       = Block( model.BUSES, rule=bus_rule )
model.line      = Block( model.LINES, rule=line_rule )
model.generator = Block( model.GENERATORS, rule=generator_rule )

def _network(model, l, end):
   if endpoint == ‘IN’:
      return model.line[l].IN == model.bus[ value(model.links[l, end]) ].PORT
   else:
      return model.line[l].OUT == model.bus[ value(model.links[l, end]) ].PORT
model.network = Constraint(model.LINES, model.ENDPOINTS, rule=_network)

def _generator_placement(model, g):
   return model.generator[g].OUT == model.bus[ value(model.generator[g].bus) ].PORT
model.generator_placement = Constraint(model.GENERATORS, rule=_generator_placement)
```

Solving GDP models

- Automated transformations generate “flat” MI(N)LPs
  - Big-M relaxation



  - Convex hull relaxation



Convert transmission model → block rule

```
def dc_economic_dispatch_rule(b, *args):
   b.bus       = Block( b.model().BUSES, rule=bus_rule )
   b.line      = Block( b.model().LINES, rule=line_rule )
   b.generator = Block( b.model().GENERATORS, rule=generator_rule )

   def _network(b, l, end):
      if endpoint == ‘IN’:
         return b.line[l].IN == b.bus[ value(b.model().links[l, end]) ].PORT
      else:
         return b.line[l].OUT == b.bus[ value(b.model().links[l, end]) ].PORT
   ed.network = Constraint(b.model().LINES, b.model().ENDPOINTS, rule=_network)

   def _gen_placement(b, g):
      return b.generator[g].OUT == b.bus[ value(b.generator[g].bus) ].PORT
   b.generator_placement = Constraint(b.model().GENERATORS, rule=_gen_placement)
```


Sandia National Laboratories

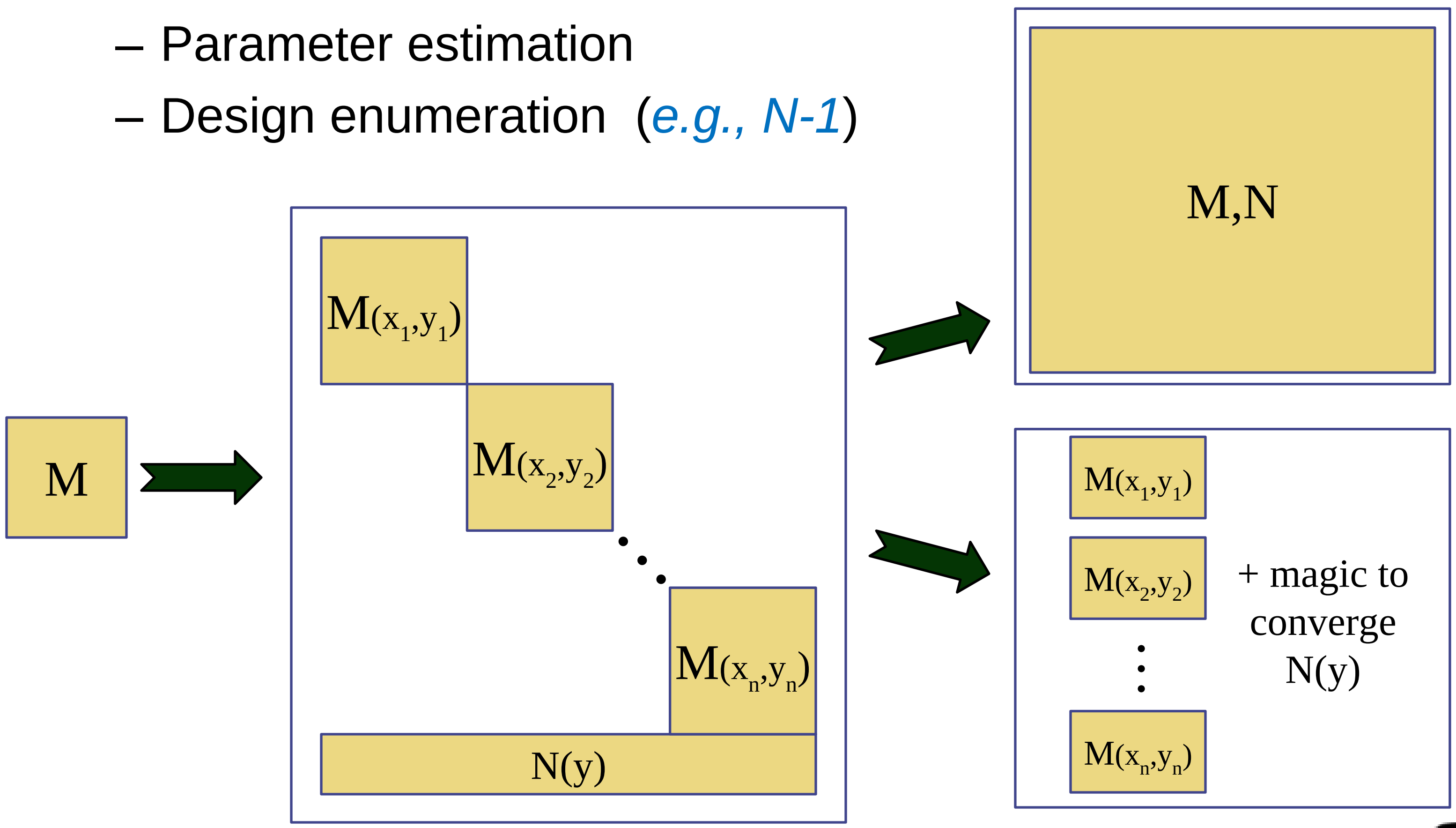
Generate the UC model

```
from power_flow import  dc_economic_dispatch_rule

model.TIMES = SET()

model.period = Block( model.TIMES, rule=dc_economic_dispatch_rule )

def _gen_limit(model, t, g):
    if t == 1:
        return Constraint.Skip
    else:
        return model.period[t-1].generator[g].STATE == \
            model.period[t].generator[g].PREV_STATE

model.generation_limit = Constraint(model.TIMES, model.GENERATORS, rule=_gen_limit)
```

Note: the generator ramp limits and startup / shutdown constraints are part of a “switchable generator” block similar to the “switchable line” block. This is a complex block (13+ parameters, 8+ variables, 7+ constraints), and is completely abstracted away by the block modeling approach.

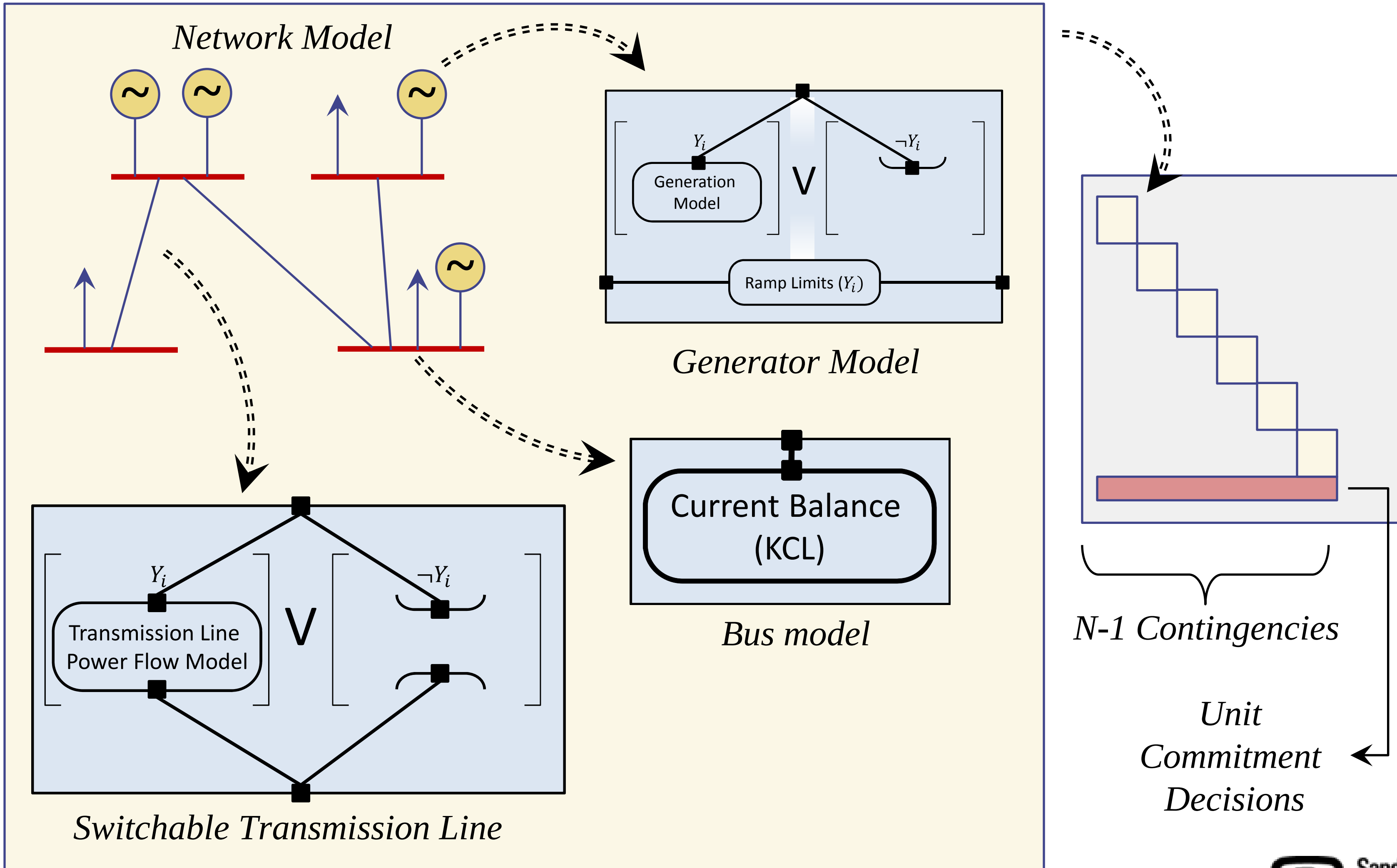
Exploiting block structure: decomposition

- “Block diagonal” models very common in optimization
  - Stochastic programming
  - Parameter estimation
  - Design enumeration (*e.g., N-1*)



Sandia National Laboratories

Putting it together: UC + switching + N-1



"Blocks" fundamentally change modeling

- Explicit model blocks
  - Component reuse
  - Implicit transformations when generating model instances
- Generalized Disjunctive Programs
  - Explicit transformations to create standard forms
  - (Solver-specific decomposition)
- Block diagonal models
  - Implicit transformation to create standard forms
  - Solver-specific decomposition

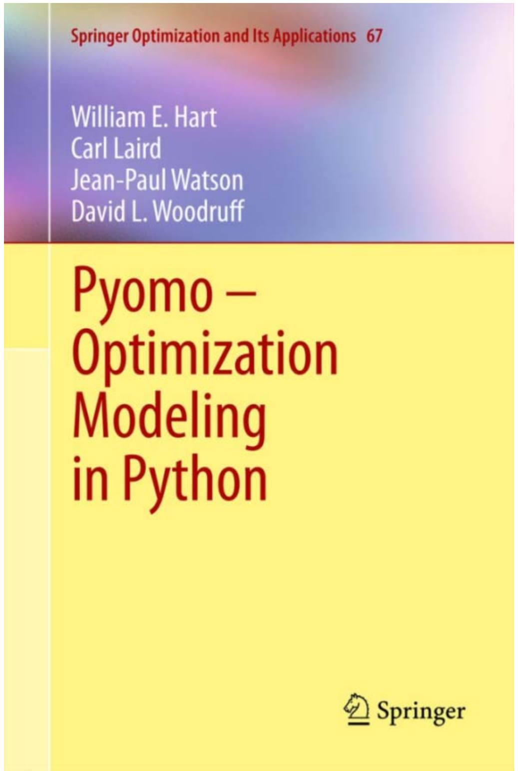
Acknowledgements

- Sandia National Laboratories
  - Bill Hart
  - Jean-Paul Watson
  - John Siirola
  - David Hart
  - Tom Brounstein
- University of California, Davis
  - Prof. David L. Woodruff
  - Prof. Roger Wets
- Texas A&M University
  - Prof. Carl D. Laird
  - Daniel Word
  - James Young
  - Gabe Hackebeil
- Texas Tech University
  - Zev Friedman
- Rose Hulman Institute
  - Tim Ekl
- William & Mary
  - Patrick Steele
- North Carolina State
  - Kevin Hunter

Plus our many users, including:
- University of California, Davis
- Texas A&M University
- University of Texas
- Rose-Hulman Institute of Technology
- University of Southern California
- George Mason University
- Iowa State University
- N.C. State University
- University of Washington
- Naval Postgraduate School
- Universidad de Santiago de Chile
- University of Pisa
- Lawrence Livermore National Lab
- Los Alamos National Lab

For more information…

- Project homepage
  - http://software.sandia.gov/coopr
- “The Book”



- Mathematical Programming Computation papers
  - Pyomo: Modeling and Solving Mathematical Programs in Python (Vol. 3, No. 3, 2011)
  - PySP: Modeling and Solving Stochastic Programs in Python (Vol. 4, No. 2, 2012)