(Very-)Large-Scale Optimization for Energy Systems: Part V: Software tools for structured (energy) optimization

Antonio Frangioni

Department of Computer Science
University of Pisa
https://www.di.unipi.it/~frangio
mailto:frangio@di.unipi.it

Scuola Nazionale Dottorandi di Elettrotecnica
Francesco Gasparini – XXVII Stage

- Part I: A View from the Stratosphere
- Part II: A Tale of (Unit) Commitment
- Part III: Handling very large formulations
- Part IV: Advanced decomposition approaches
- Part V: Software tools for structured (energy) optimization

Outline – Part V

Decomposition-aware modelling systems and solvers

SMS++ : design goals

SMS++ : basic ideas

Many other necessary pieces

SMS++ : existing `Block` and `Solver`

More advanced `Block` / `Solver`

SMS++ and beyond: (some of) the missing pieces

Conclusions (for good)

- Most interactions with optimization solvers via Algebraic Modelling Languages (AML): commercial `AMPL`[1], `GAMS`[2], `AIMMS`[3] and `OPL`[4], or open-source `Coliop`[5] and `ZIMPL`[6]
- Interfaced with a varying set (few/many) of general-purpose solvers for large problem classes (MILP, MINLP, conic, . . .)
- AML is a separate language, typically interpreted (not efficient)
- Mostly "flat" languages (no OOP), modularity an issue
- Focus on "model once, solve once"; some offer some support for iterative procedures but clearly an afterthought
- Hide the complexities of the model / solution process to inexperienced users

---

[1] https://ampl.com
[2] https://www.gams.com
[3] https://www.aimms.com/platform/aimms-development
[4] https://www.ibm.com/docs/en/icos/12.8.0.0?topic=opl-optimization-programming-language
[5] http://www.coliop.org
[6] https://zimpl.zib.de

Modelling systems, and what they are for

- Modelling systems: libraries written in general-purpose languages providing similar functionalities to AML
- Typically open-source: `FLOPCpp`[7], `COIN Rehearse`[8], `Gravity`[9] (`C++`), `PuLP`[10], `Pyomo`[11] (`Python`), `JuMP`[12] (`Julia`), `YALMIP`[13] (`Matlab`)
- May not fully replicate AML constructs, sometimes more limited
- Solver interfacing and overhead lower with efficient languages (`C++`)
- Multiple models and iterative procedures more natural
- Can exploit OOP features of host language for better modularity
- Mostly focus on general-purpose solvers and "model once, solve once"
- Tailored for end-users, not algorithms developers

---

[7] https://github.com/coin-or/FlopCpp
[8] https://github.com/coin-or/Rehearse
[9] https://github.com/coin-or/Gravity
[10] https://github.com/coin-or/pulp
[11] http://www.pyomo.org
[12] https://github.com/jump-dev/JuMP.jl
[13] https://yalmip.github.io

- Some solvers provide decomposition capabilities:
  - Cplex[14] does Benders', structure automatic or user hints
  - SCIP[15] does B&C&P (one-level D-W), pricing & reformulation up to the user (plugins)
  - GCG[16] extends SCIP with automatic and user-defined generic D-W and Benders' (one-level, Marco says multi-level possible but under debug)
  - DDSIP[17] and PIPS[18] implement D-W for two-stage stochastic programs
  - The BaPCoD B&C&P code has been used to develop Coluna.jl[19], doing one-level D-W and (alpha) Benders', multi-level planned
- Other solvers use structure in different ways: BlockIP[20], OOPS[21] (structured interior-point methods[22], another interesting idea)

---

[14] https://www.ibm.com/products/ilog-cplex-optimization-studio
[15] https://scipopt.org
[16] https://gcg.or.rwth-aachen.de
[17] https://github.com/RalfGollmer/ddsip
[18] https://github.com/Argonne-National-Laboratory/PIPS
[19] https://github.com/atoptima/Coluna.jl
[20] http://www-eio.upc.edu/~jcastro/BlockIP.html
[21] https://www.maths.ed.ac.uk/~gondzio/parallel/solver.html
[22] F., Gentile "New Preconditioners for KKT Systems of Network Flow Problems" *SIOPT* 2004

- In a word?

[23] https://www.maths.ed.ac.uk/ERGO/sml

[24] https://github.com/StructJuMP/StructJuMP.jl

[25] https://github.com/atoptima/BlockDecomposition.jl

- In a word? Very few (that's two words . . .)
- `OOPS` is interfaced with `SML`[23], providing some parallel capabilities
- `PIPS` is interfaced with `StructJuMP`[24], using `BlockDecomposition`[25]

[23] https://www.maths.ed.ac.uk/ERGO/sml

[24] https://github.com/StructJuMP/StructJuMP.jl

[25] https://github.com/atoptima/BlockDecomposition.jl

- In a word? Very few (that's two words . . .)
- `OOPS` is interfaced with `SML`[23], providing some parallel capabilities
- `PIPS` is interfaced with `StructJuMP`[24], using `BlockDecomposition`[25]
- No modelling system is focused on multi-level structure, non-general-purpose solvers, parallel, and modularity/extendability
- Although `JuMP` is doing a good job at promoting some of these

[23] https://www.maths.ed.ac.uk/ERGO/sml

[24] https://github.com/StructJuMP/StructJuMP.jl

[25] https://github.com/atoptima/BlockDecomposition.jl

- In a word? Very few (that's two words . . .)
- `OOPS` is interfaced with `SML`[23], providing some parallel capabilities
- `PIPS` is interfaced with `StructJuMP`[24], using `BlockDecomposition`[25]
- No modelling system is focused on multi-level structure, non-general-purpose solvers, parallel, and modularity/extendability
- Although `JuMP` is doing a good job at promoting some of these
- We tried working with `Julia`, but most solvers are in `C` / `C++`, and the full circle `Julia` → `C++` → `Julia` did not work well
- So we choose no-performance-compromise `C++`, accepting the drawbacks

---

[23] https://www.maths.ed.ac.uk/ERGO/sml

[24] https://github.com/StructJuMP/StructJuMP.jl

[25] https://github.com/atoptima/BlockDecomposition.jl

Outline

Decomposition-aware modelling systems and solvers

SMS++ : design goals

SMS++ : basic ideas

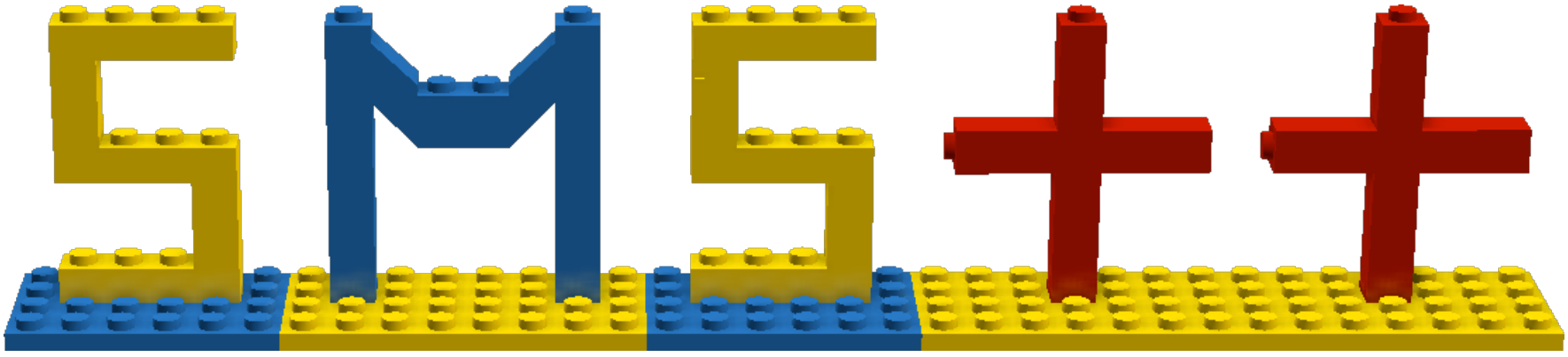
Many other necessary pieces

SMS++ : existing Block and Solver

More advanced Block / Solver

SMS++ and beyond: (some of) the missing pieces

Conclusions (for good)



https://gitlab.com/smspp/smspp-project

"For algorithm developers, from algorithm developers"

Open source (LGPL3)

Version 0.6.0 as of December 2025, still a long way to go

What SMS++ is

- A core set of C++-20 classes implementing a modelling system that:
  - explicitly supports the notion of `Block` $\equiv$ nested structure
  - separately provides "semantic" information from "syntactic" details
    (list of constraints/variables $\equiv$ one specific formulation among many)
  - allows exploiting specialised `Solver` on `Block` with specific structure
  - manages any dynamic change in the `Block`
    beyond "just" generation of constraints/variables
  - supports reformulation/restriction/relaxation of `Block`
  - has built-in parallel processing capabilities
  - should be able to deal with almost anything (bilevel, PDE, . . .)
- An hopefully growing set of specialized `Block` and `Solver`
- In perspective an ecosystem fostering collaboration and code sharing:
  a community-building effort as much as a (suite of) software product(s)

What SMS++ is not

- An algebraic modelling language: `Block` / `Solver` are C++ code
  (although it provides some modelling-language-like functionalities)
- For the faint of heart: primarily written for algorithmic experts
  (although users may benefit from having many pre-defined `Block`)
- Stable: only version 0.6.0, lots of further development ahead,
  significant changes in interfaces not ruled out, actually expected
  (although current `Block` / `Solver` very thoroughly tested)
- Interfaced with many existing solvers: `Cplex`, `SCIP`, `Gurobi`, `HiGHS`, `MCFClass`, `LEMON`, `StOpt`
  (although the list should hopefully grow, and there are built-in ones)

Outline

Decomposition-aware modelling systems and solvers

SMS++ : design goals

SMS++ : basic ideas

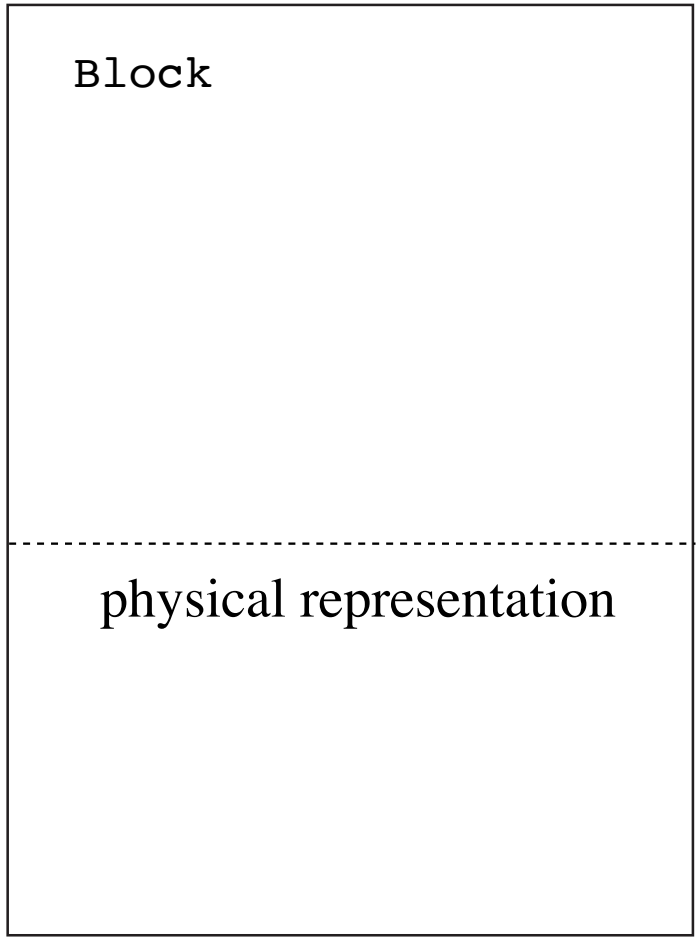
Many other necessary pieces

SMS++ : existing Block and Solver

More advanced Block / Solver

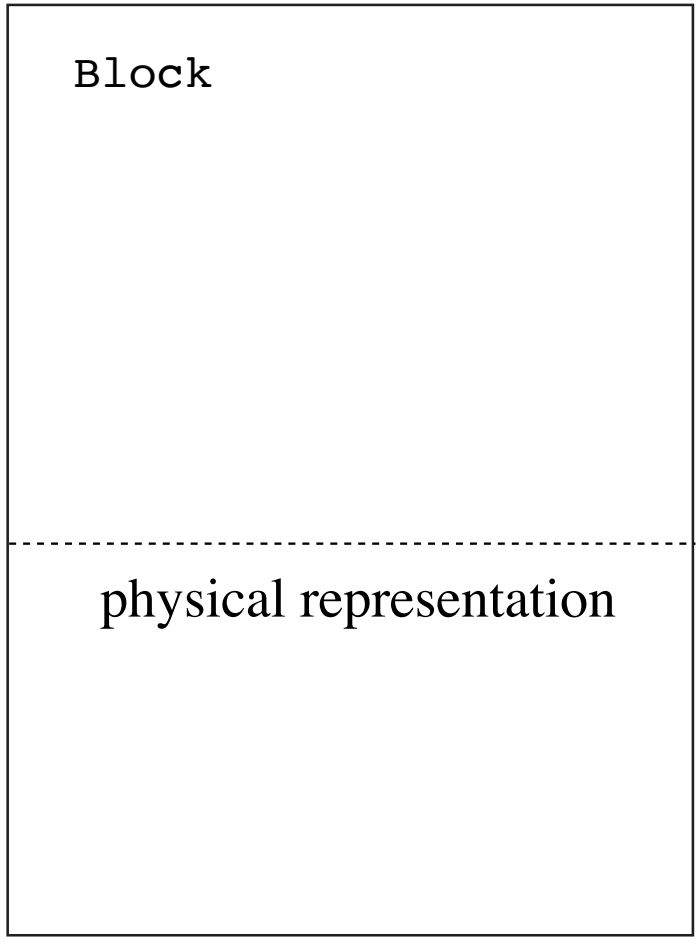
SMS++ and beyond: (some of) the missing pieces

Conclusions (for good)



- For a man with a hammer everything is a nail

It starts deceivingly simple



- For a man with a solver everything is a `Block` (call me blockhead )
- `Block` = abstract class representing the general concept of "(fragment of) mathematical model with a well-understood semantic"

It starts deceivingly simple (flow)

`MCFBlock : Block`

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij}$$

$$\sum_{(i,j)\in FS(i)} x_{ij} - \sum_{(j,i)\in BS(i)} x_{ji} = b_i \quad i \in N$$

$$0 \le x_{ij} \le u_{ij} \qquad (i,j) \in A$$

```
n    m
SN[] EN[] C[] U[] B[]
```

- For a man with a solver everything is a `Block` (call me blockhead )
- `Block` = abstract class representing the general concept of "(fragment of) mathematical model with a well-understood semantic"
- Each `:Block` a model with specific structure:
  `MCFBlock : Block` = a Min-Cost Flow problem

It starts deceivingly simple (1UC)

`ThermalUnitBlock : Block`

$$\min \sum_{i\in P} c^i(\mathbf{p}^i,\mathbf{u}^i) = \sum_{i\in P}\left(c^i(\mathbf{u}^i) + \sum_{t\in\mathscr{T}} c_t^i(p_t^i)\right)$$

$$\bar{p}^i_{\min} u^i_t \leqslant p^i_t \leqslant \bar{p}^i_{\max} u^i_t \quad t\in\mathscr{T}$$

$$p^i_t \leqslant p^i_{t-1} + u^i_{t-1}\Delta^i_+ + (1-u^i_{t-1})\bar{l}^i \quad t\in\mathscr{T}$$

$$p^i_{t-1} \leqslant p^i_t + u^i_t\Delta^i_- + (1-u^i_t)\bar{u}^i \quad t\in\mathscr{T}$$

$$u^i_t \geqslant u^i_r - u^i_{r-1} \quad t\in\mathscr{T},\quad r\in[t-\tau^i_+, t-1]$$

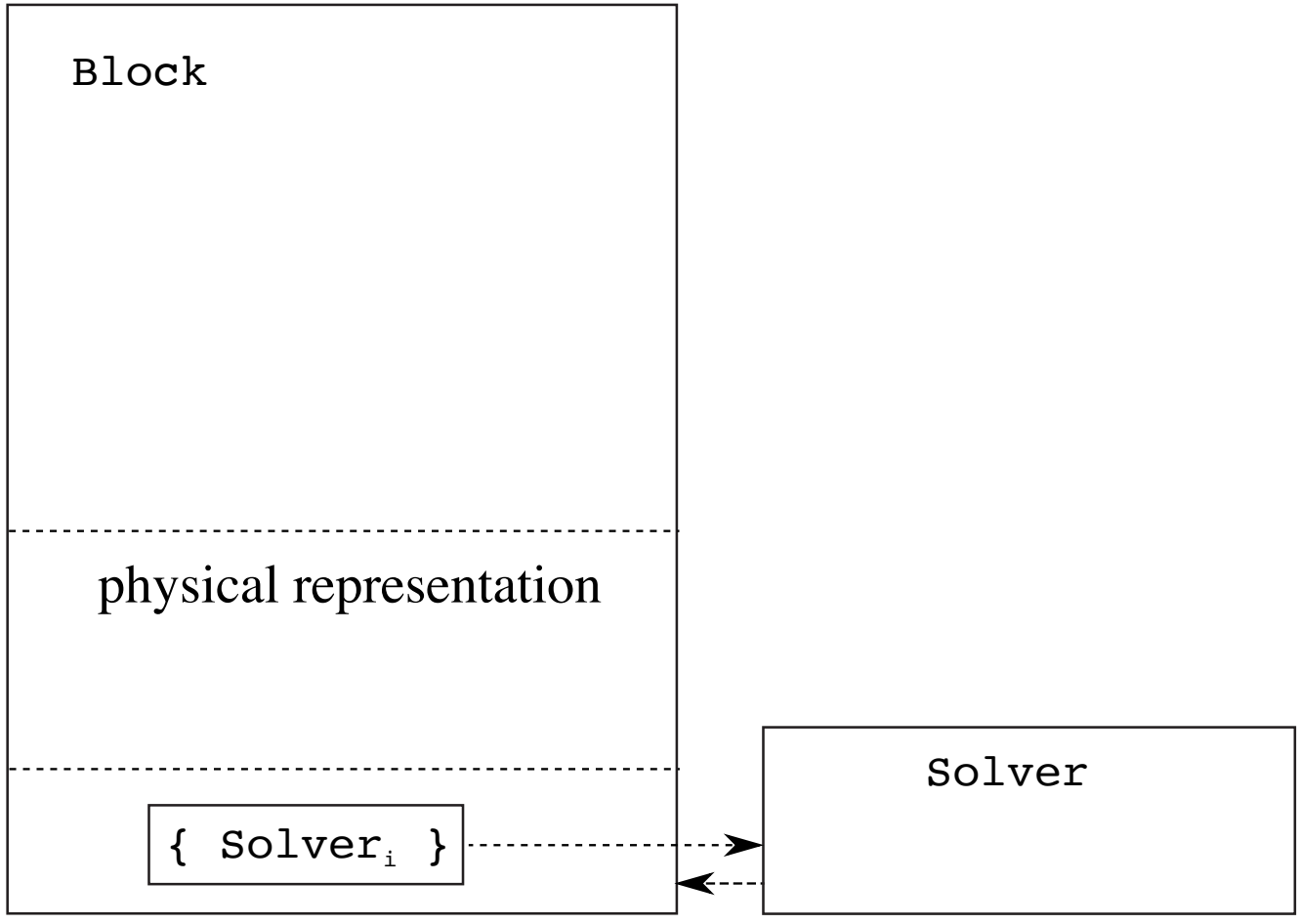
$$u^i_t \geqslant 1 - u^i_{r-1} - u^i_r \quad t\in\mathscr{T},\quad r\in[t-\tau^i_-, t-1]$$

$$u^i_t \in \{0,1\} \quad t\in\mathscr{T}$$

```
T   a[] b[] c[]   Dp[]
p_min[] p_max[] Dm[]
```

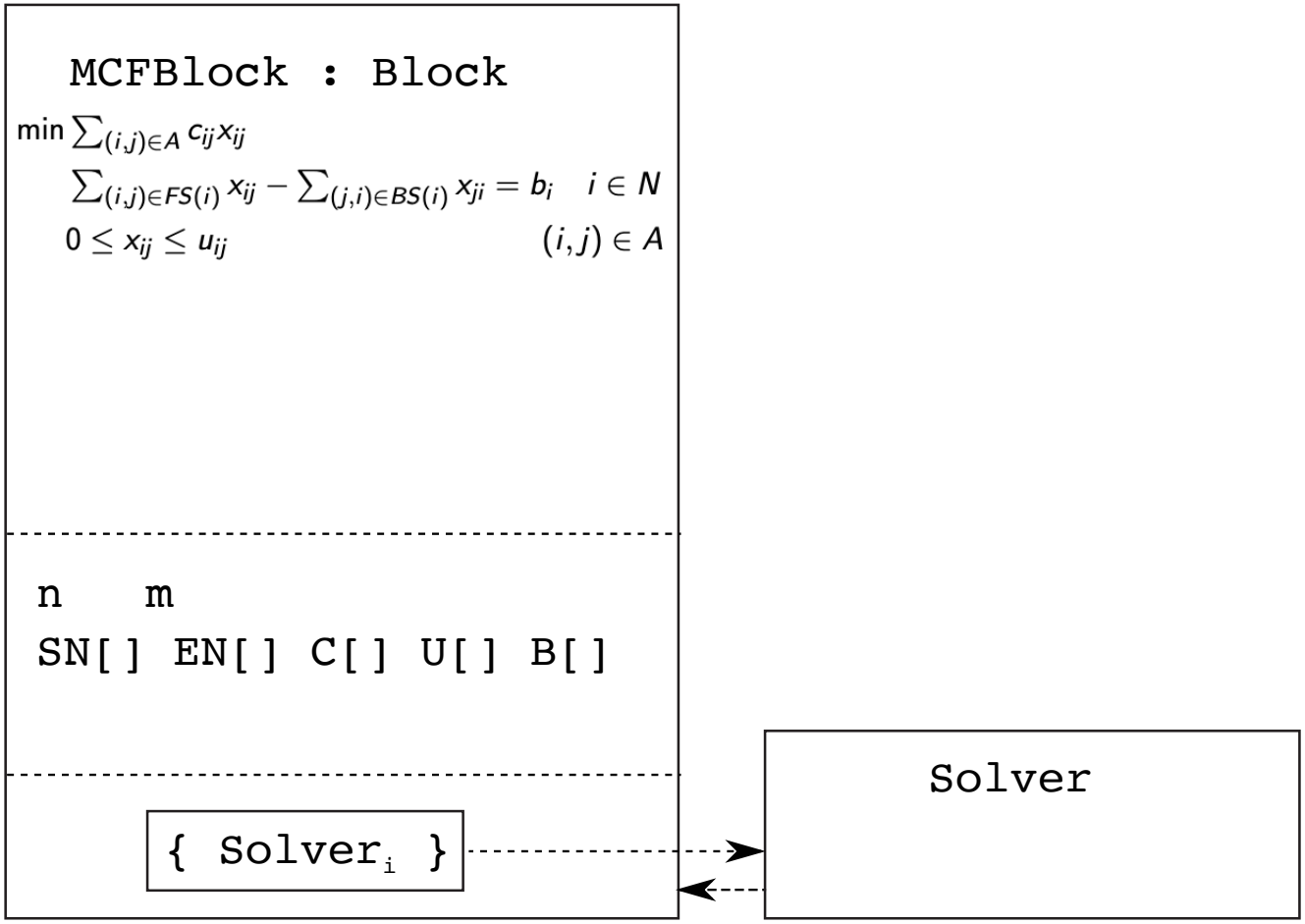
- For a man with a solver everything is a `Block` (call me blockhead )
- `Block` = abstract class representing the general concept of "(fragment of) mathematical model with a well-understood semantic"
- Each `:Block` a model with specific structure:
  `ThermalUnit : Block` = a single-(thermal)-Unit Commitment problem

A `Block` ∃ (≈) because a (specialised) `Solver` ∃



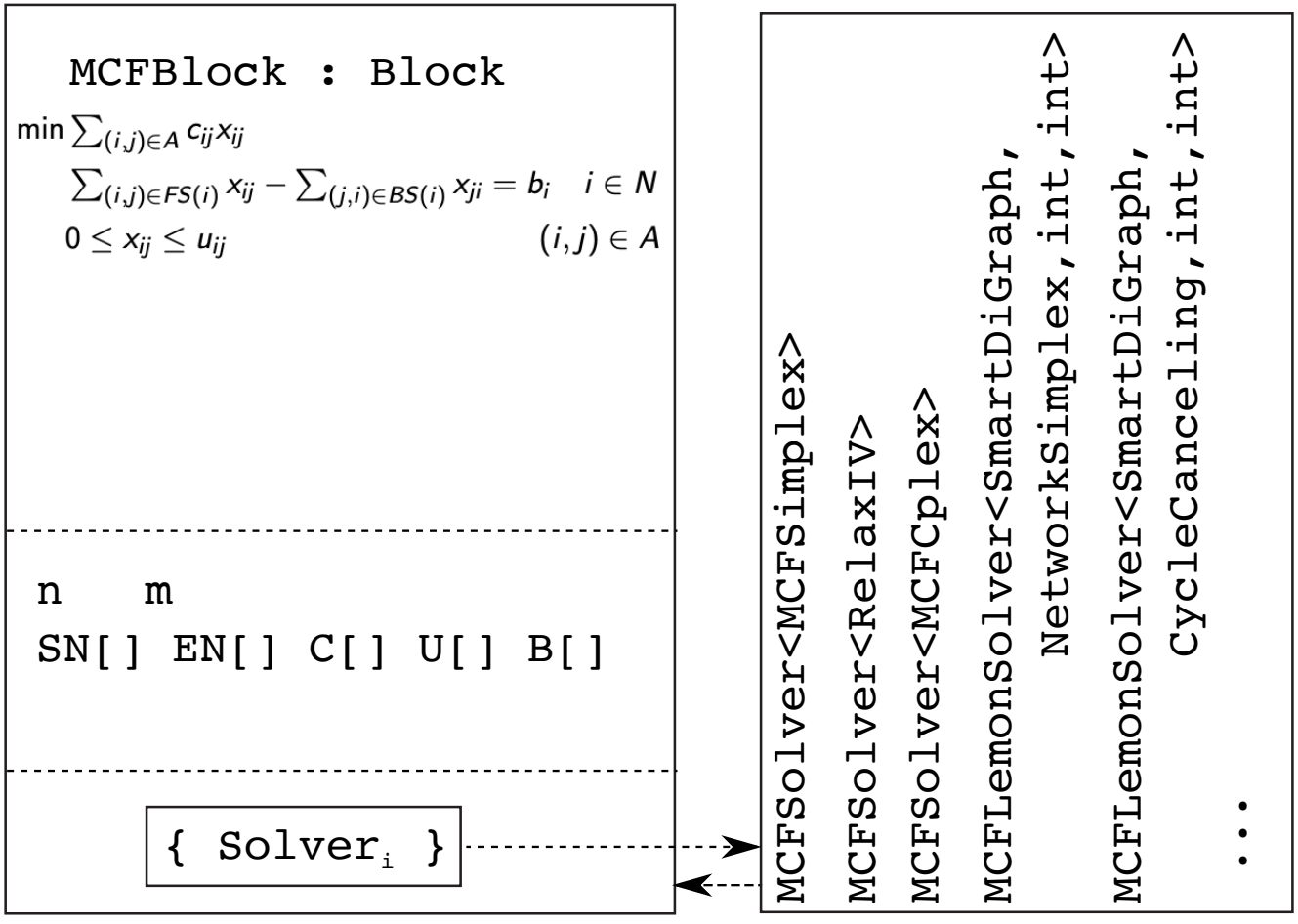
- Any number of `Solver` can be attached to a `Block`

A `Block` ∃ (≈) because a (specialised) `Solver` ∃ (flow)



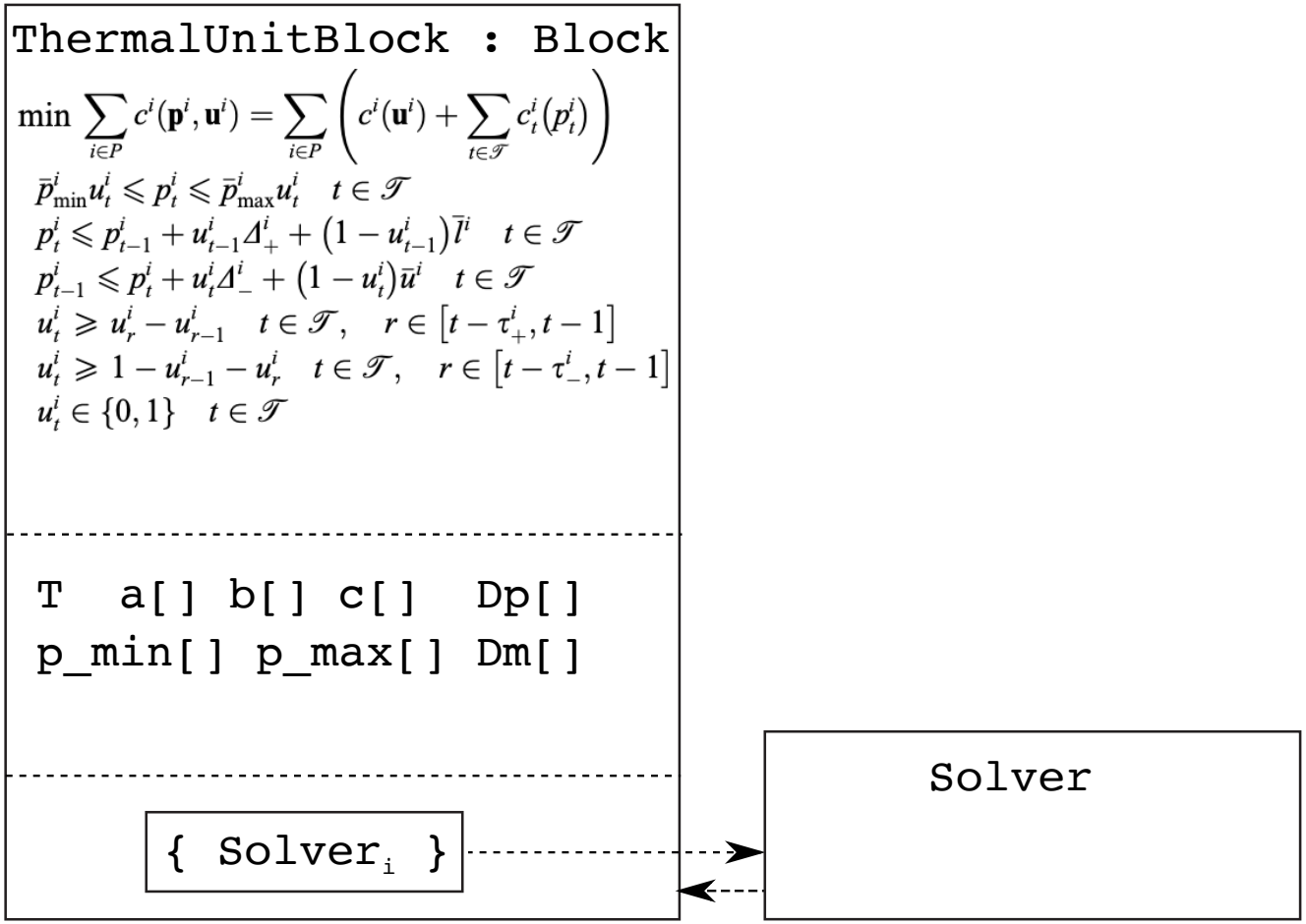
- Any number of `Solver` can be attached to a `Block`
- Any specific `:Block` (e.g., `MCFBlock`) can have specialised `:Solver`
- Only work for that `:Block` but being specialised they can be fast
- Can be

A `Block` ∃ (≈) because a (specialised) `Solver` ∃ (flow)



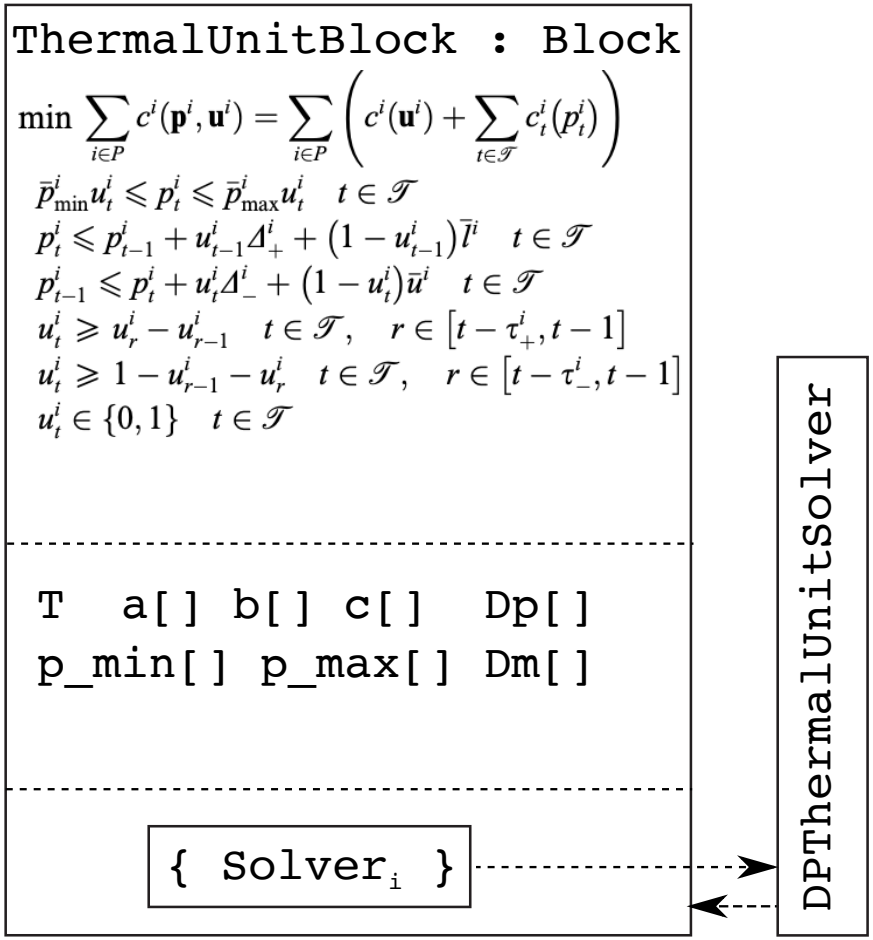
- Any number of `Solver` can be attached to a `Block`
- Any specific `:Block` (e.g., `MCFBlock`) can have specialised `:Solver`
- Only work for that `:Block` but being specialised they can be fast
- Can be wrapper classes to efficient existing (C++) libraries

A `Block` ∃ (≈) because a (specialised) `Solver` ∃ (1UC)



- Any number of `Solver` can be attached to a `Block`
- Any specific `:Block` (e.g., `ThermalUnitBlock`) can have specialised `:Solver`
- Only work for that `:Block` but being specialised they can be fast
- Can be

A `Block` ∃ (≈) because a (specialised) `Solver` ∃ (1UC)



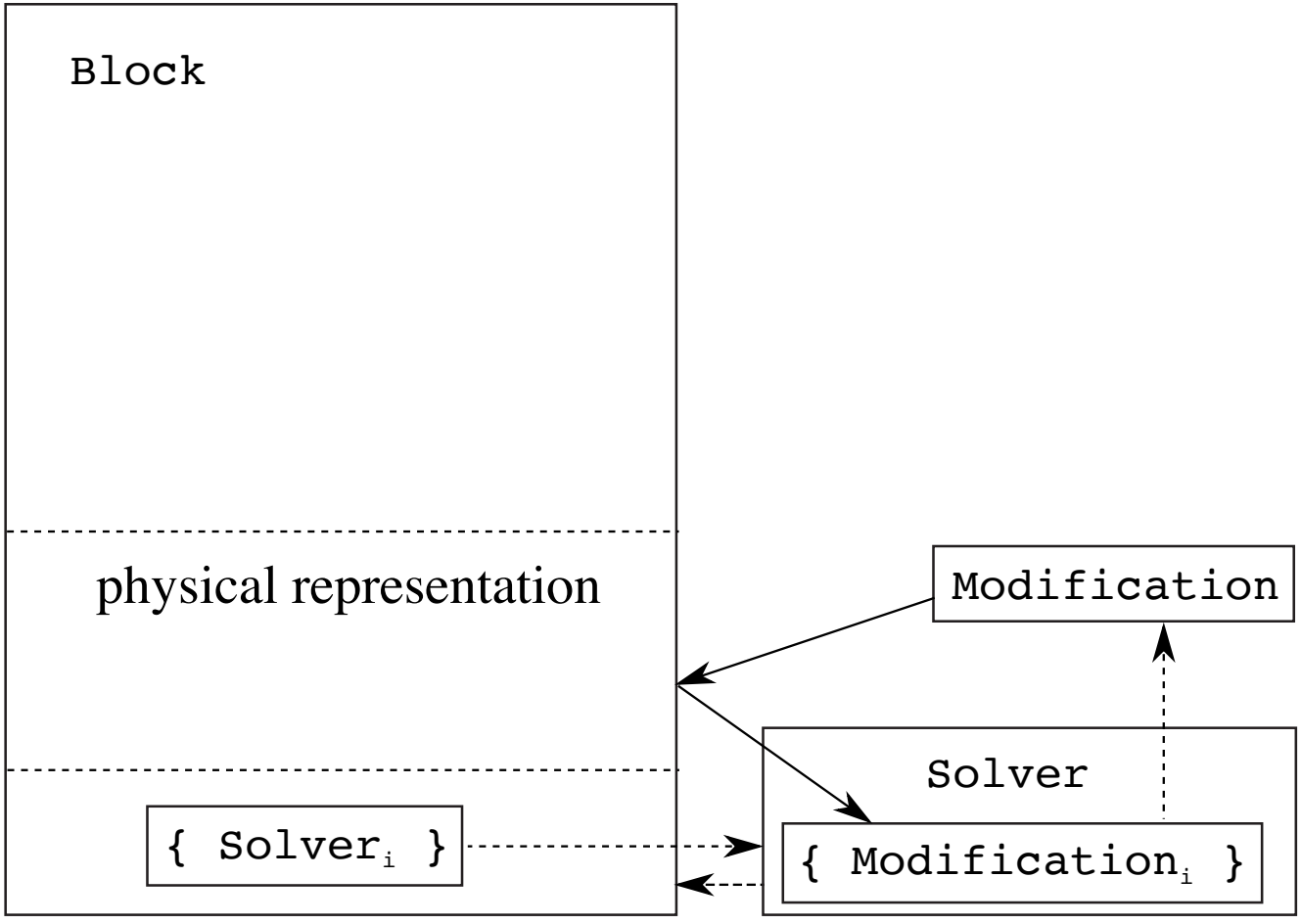
- Any number of `Solver` can be attached to a `Block`
- Any specific `:Block` (e.g., `ThermalUnitBlock`) can have specialised `:Solver`
- Only work for that `:Block` but being specialised they can be fast
- Can be native `SMS++` implementations[26]

[26] F., Gentile "Solving Nonlinear Single-Unit Commitment Problems with Ramping Constraints" *O.R.* 2006

Very important: all `Solver` look the same

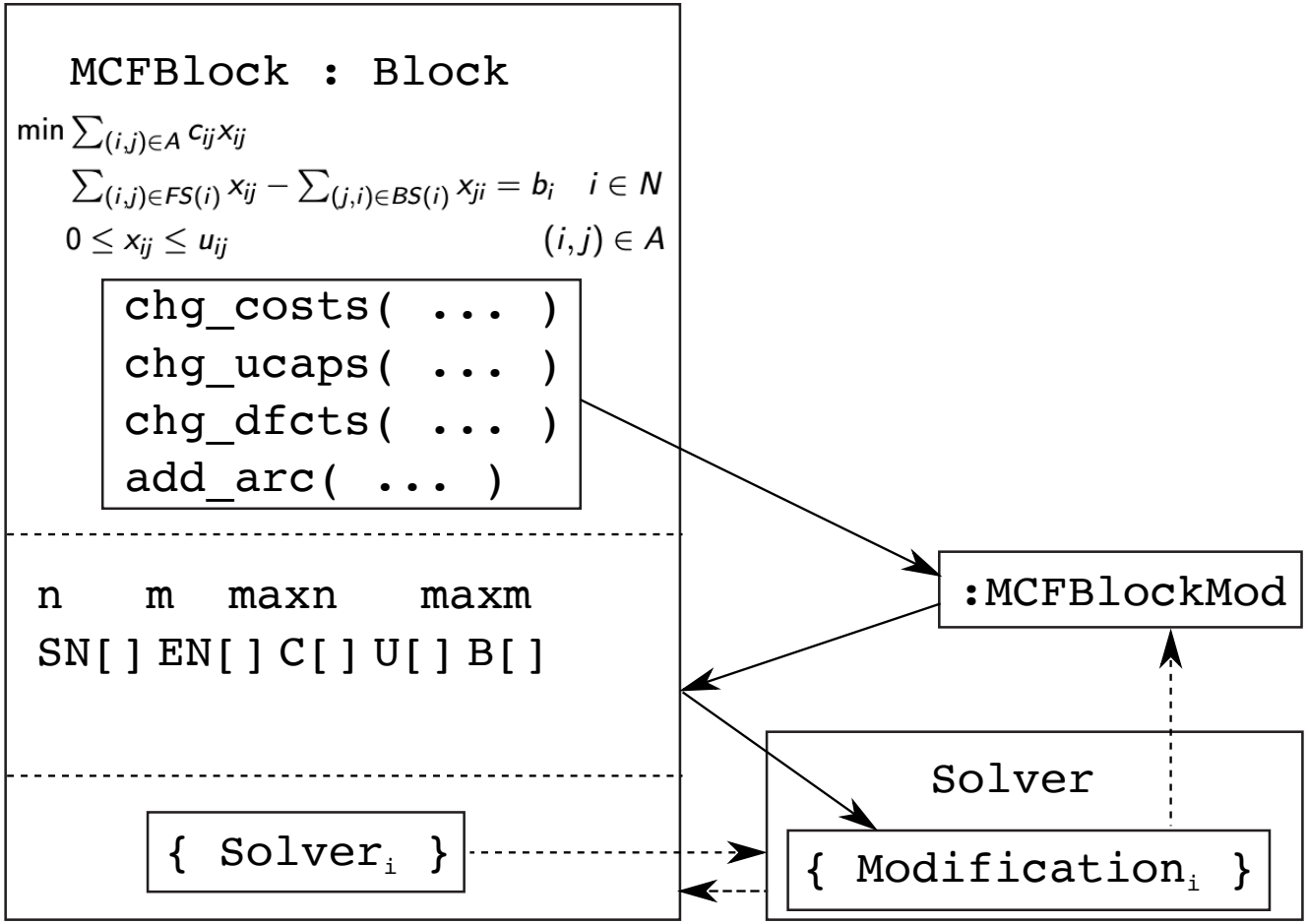
- Input / output of `Solver` is via `Block`: minimal and uniform interface, solutions written into the `Variable` of the `Block` (will change)
- `Solver : ThinComputeInterface`: "anything that can be computed"
- Uniform provisions for the fact that `compute()` can:
  - end in several ways (OK, error, stopped, ...) providing on demand any # of (≈) optimal solutions or unboundedness / unfeasibility certificates
  - be influenced by `vector<int / double / string>` parameters which can be gathered in generic `ComputeConfig : Configuration` object
  - standard time / resource / threads limits, but restarts (reoptimization)
  - lazily reacts to changes in the data of the `Block` via `Modification`
  - periodically invoke user-defined events
- `CDASolver : Solver`: bounds are associated to (multiple) dual solutions
- `compute_async()` using `std::future` for (thread-based) parallelization
- `Solver` factory allows to create any `Solver` out of `string` (classname)
- Switching to a different `Solver` is immediate (modulo `ComputeConfig`)

Solving a `Block` is (almost) never the endgame



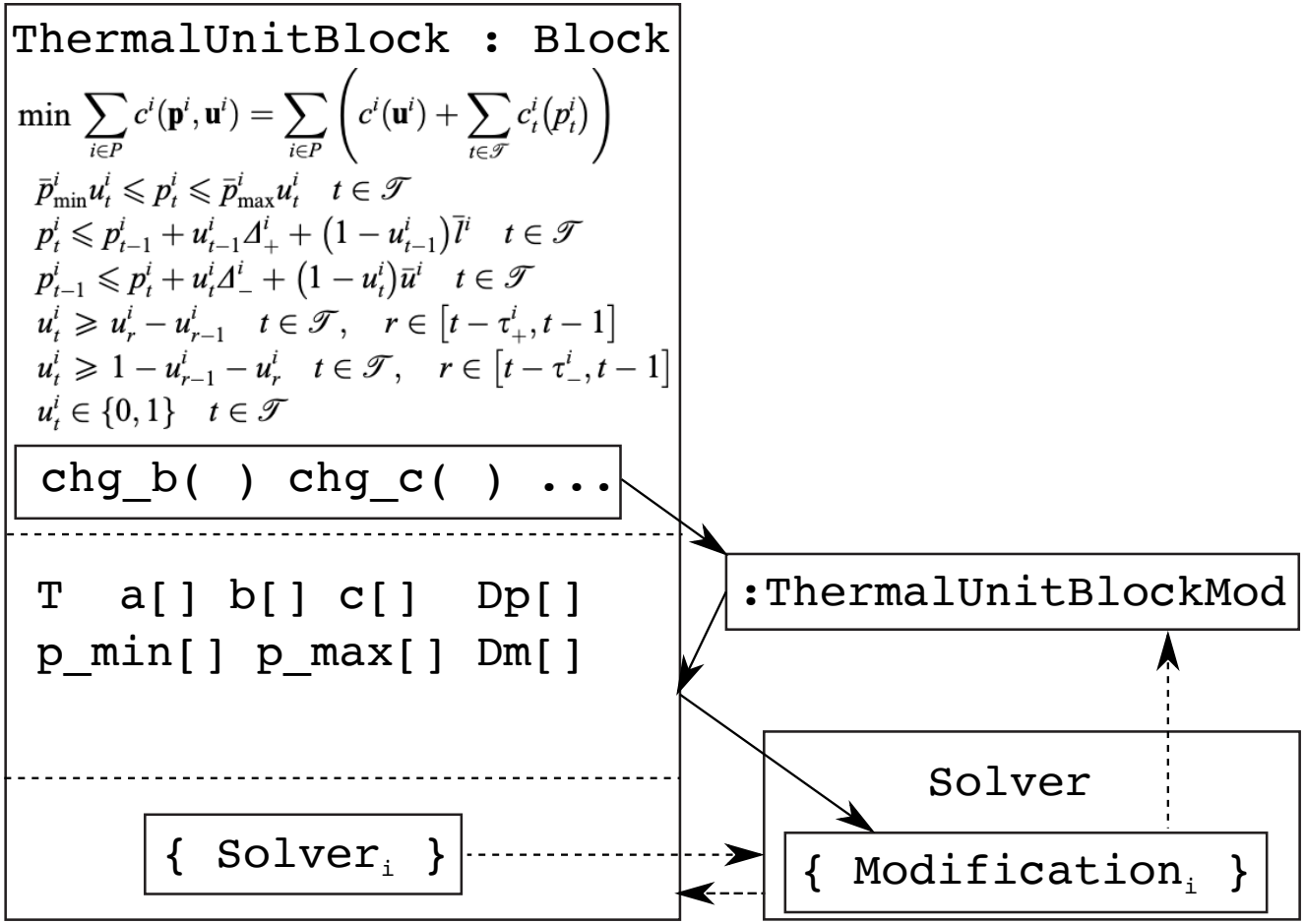
- No "Après moi, le déluge": a `:Block` will (hopefully) have many uses
- Data in the `:Block` will change, `Solver` receive `Modification` about it
- `list<shared_ptr<Modification>>` automatically updated in all `Solver`

Solving a `Block` is (almost) never the endgame (flow)



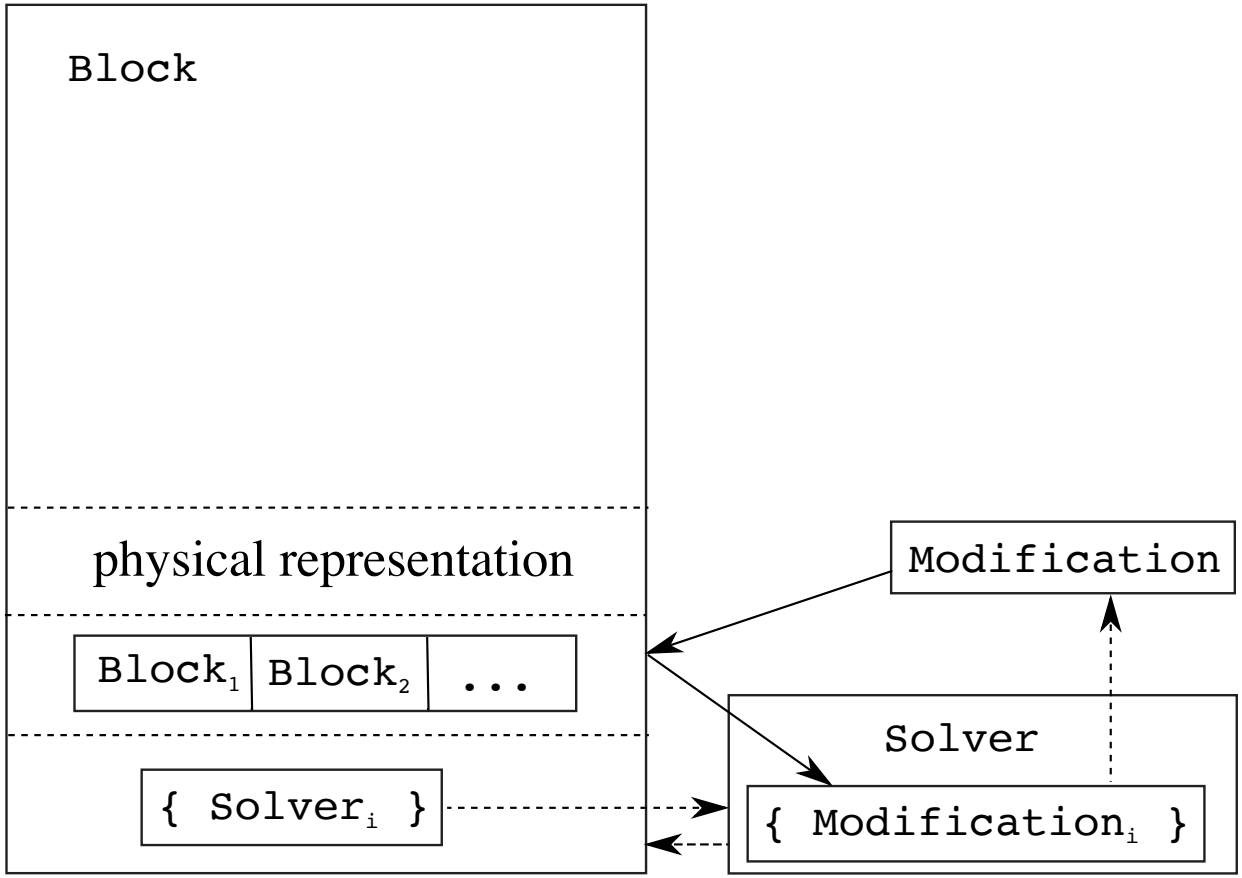
- No "Après moi, le déluge": a `:Block` will (hopefully) have many uses
- Data in the `:Block` will change, `Solver` receive `Modification` about it
- `list<shared_ptr<Modification>>` automatically updated in all `Solver`
- Each `:Block` decides what / how change, issue specialised `:Modification`

Solving a `Block` is (almost) never the endgame (1UC)



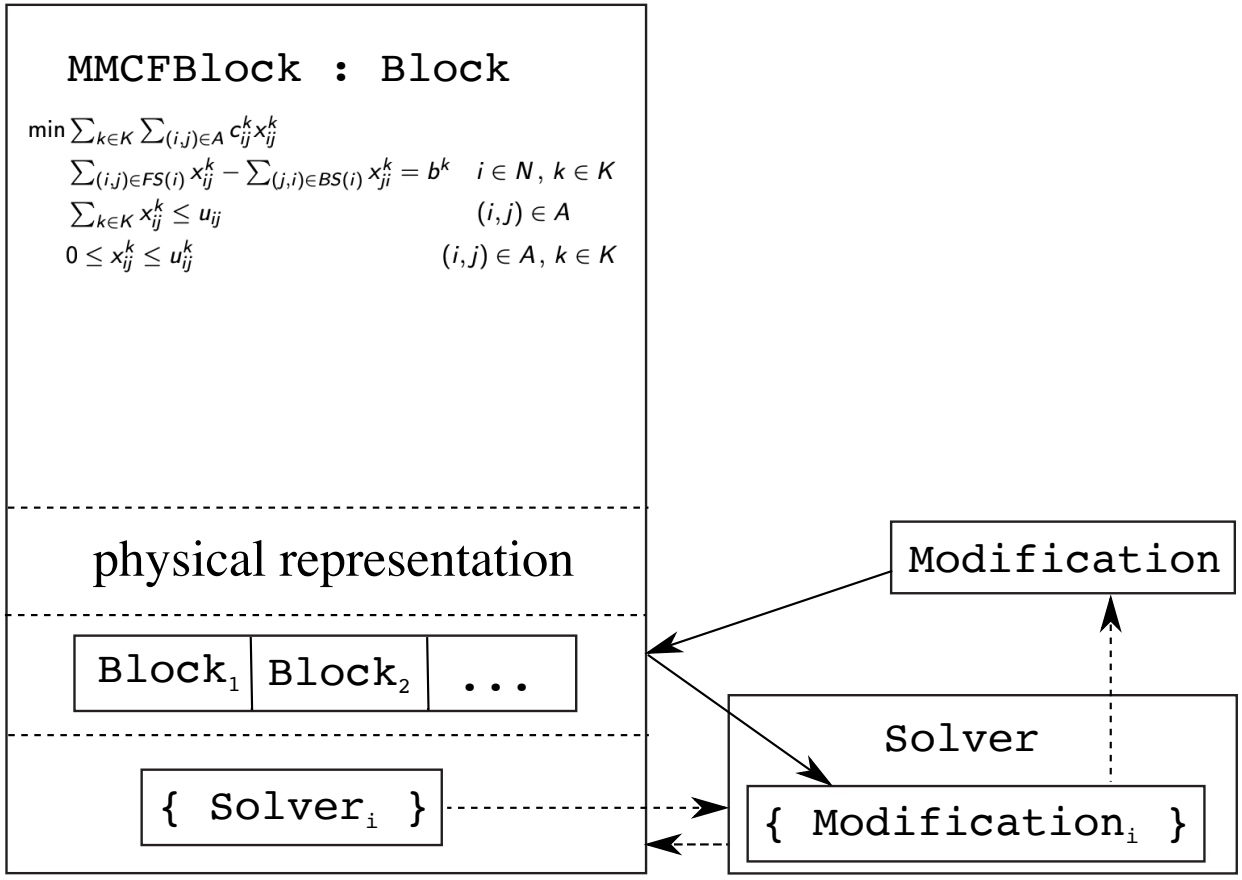
- No "Après moi, le déluge": a `:Block` will (hopefully) have many uses
- Data in the `:Block` will change, `Solver` receive `Modification` about it
- `list<shared_ptr<Modification>>` automatically updated in all `Solver`
- Each `:Block` decides what / how change, issue specialised `:Modification`

A `Block` is (almost) always just a (small) part



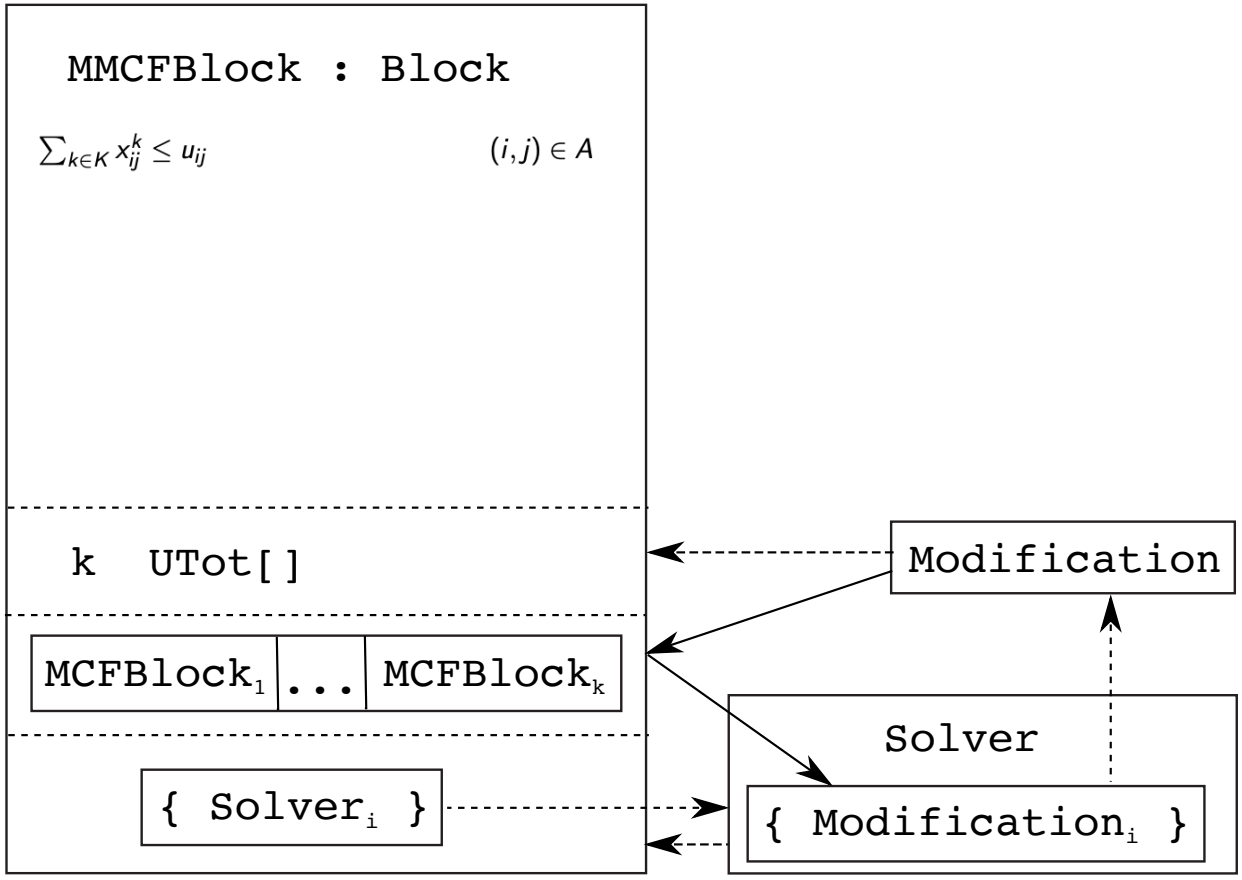
- A `Block` can have any # of sub-`Block`, recursively (`Block *`);
- Problem data split between them
- `Modification` sent to all `Solver` attached to `Block` and its ancestors
- `load()`-ed from text files or `[de]serialize()`-d to / from `netCDF` files

A `Block` is (almost) always just a (small) part (multiflow)



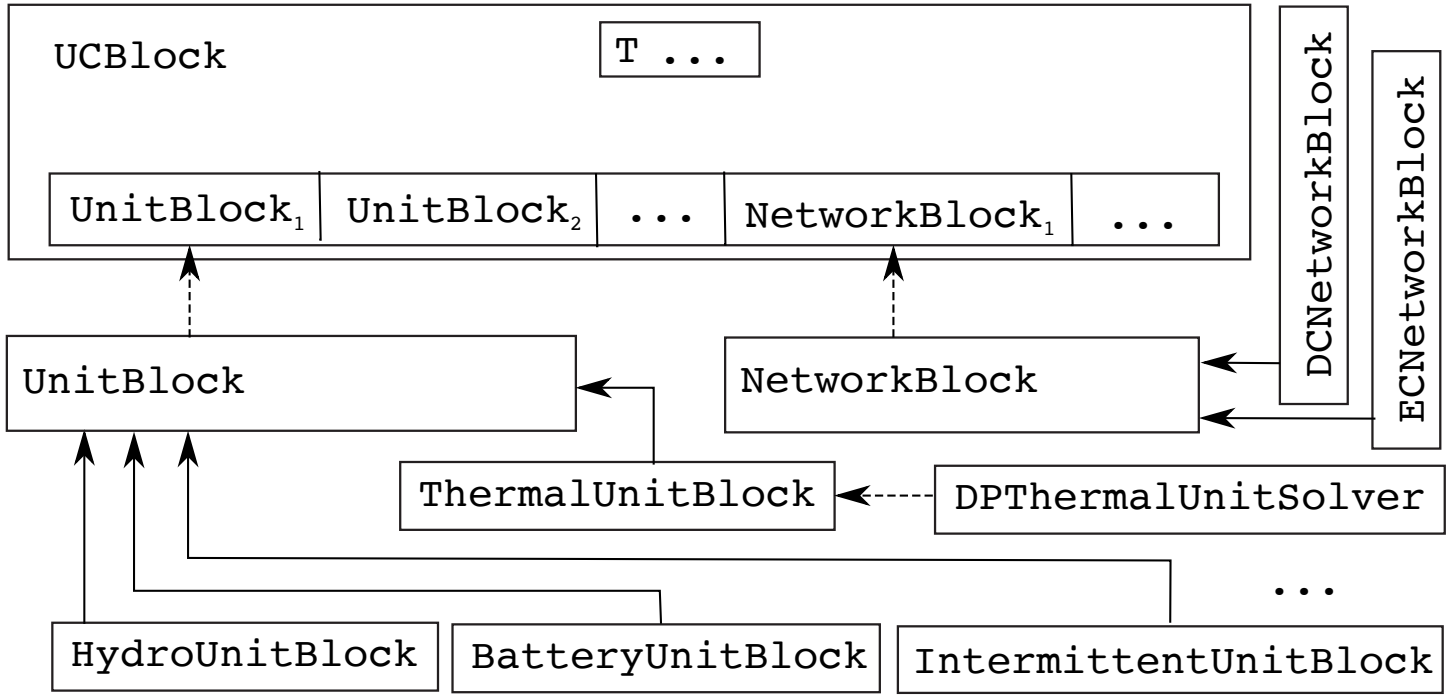
- A `Block` can have any # of sub-`Block`, recursively (`Block *`); e.g., `MMCFBlock : Block` has k sub-`MCFBlock`
- Problem data split between them
- `Modification` sent to all `Solver` attached to `Block` and its ancestors
- `load()`-ed from text files or `[de]serialize()`-d to / from `netCDF` files

A `Block` is (almost) always just a (small) part (multiflow)



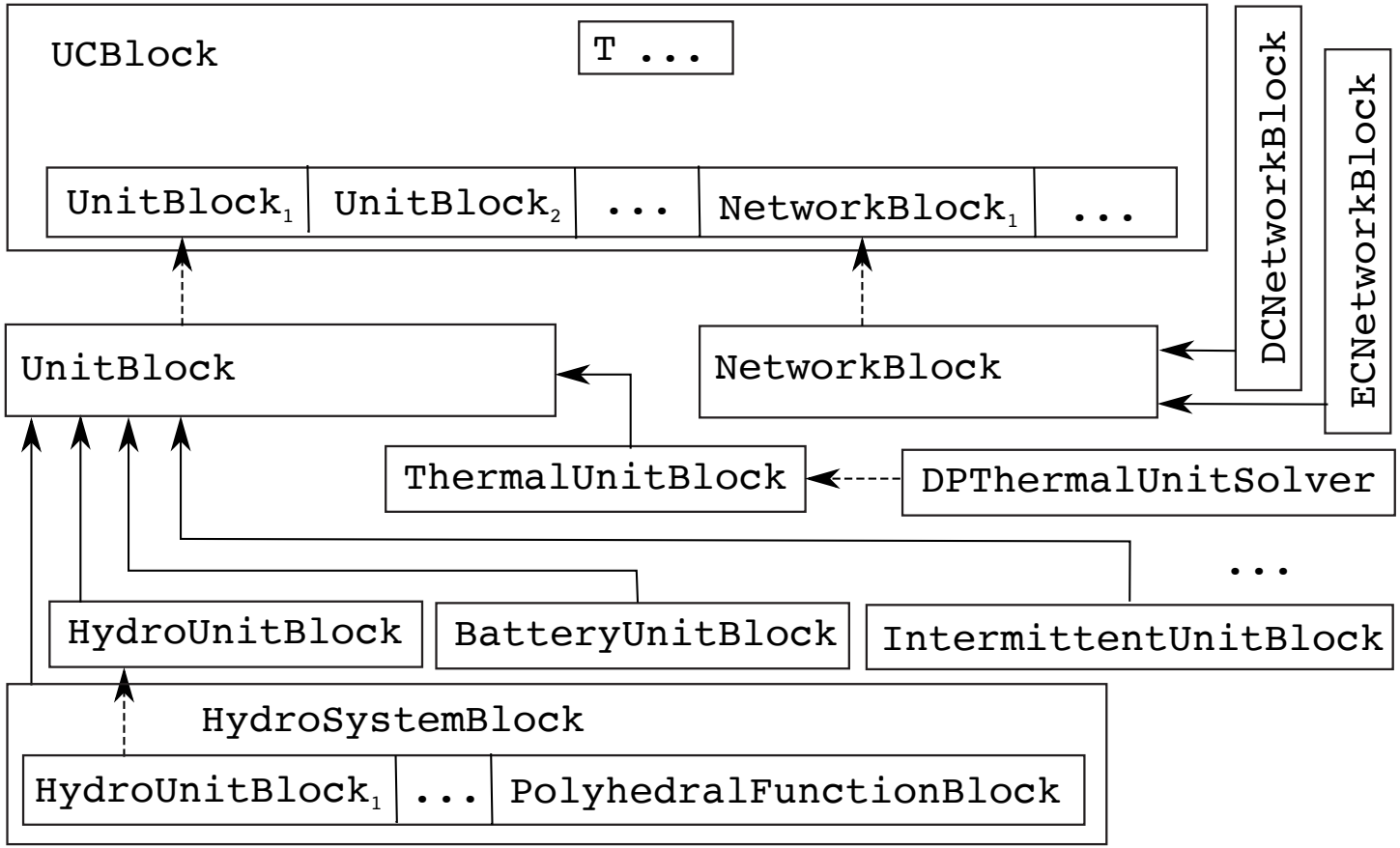
- A `Block` can have any # of sub-`Block`, recursively (`Block *`); e.g., `MMCFBlock : Block` has k sub-`MCFBlock`
- Problem data split between them: u_{ij} only in `MMCFBlock`
- `Modification` sent to all `Solver` attached to `Block` and its ancestors
- `load()`-ed from text files or `[de]serialize()`-d to / from `netCDF` files

A `Block` is (almost) always just a (small) part (UC)



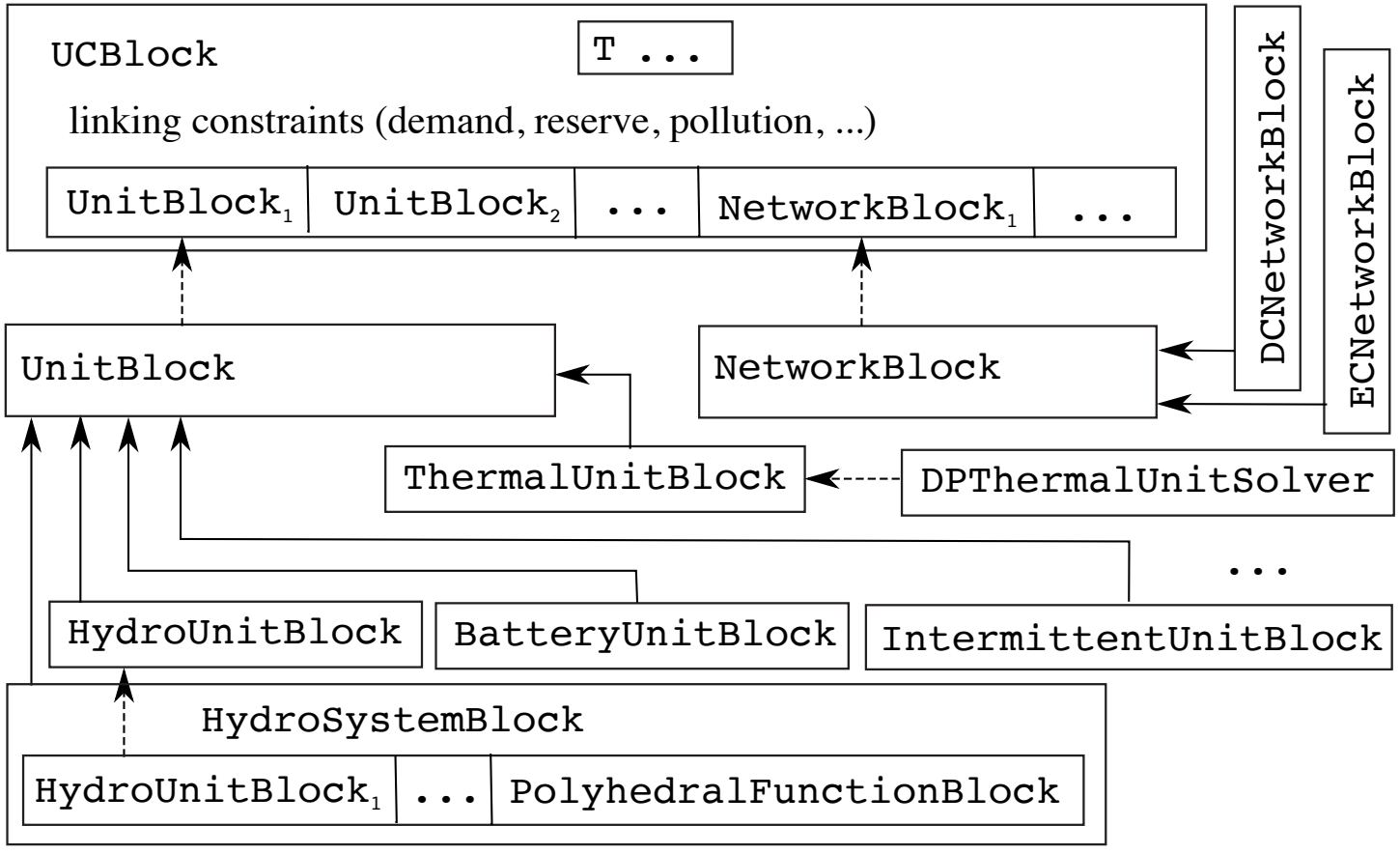
- A `Block` can have any # of sub-`Block`, recursively (`Block *`); e.g., `UCBlock : Block` has k `:UnitBlock` and T `:NetworkBlock`
- Problem data split between them:
- `Modification` sent to all `Solver` attached to `Block` and its ancestors
- `load()`-ed from text files or `[de]serialize()`-d to / from `netCDF` files

A `Block` is (almost) always just a (small) part (UC)



- A `Block` can have any # of sub-`Block`, recursively (`Block *`); e.g., `UCBlock : Block` has k `:UnitBlock` and T `:NetworkBlock` recursively
- Problem data split between them:
- `Modification` sent to all `Solver` attached to `Block` and its ancestors
- `load()`-ed from text files or `[de]serialize()`-d to / from `netCDF` files

A `Block` is (almost) always just a (small) part (UC)



- A `Block` can have any # of sub-`Block`, recursively (`Block *`); e.g., `UCBlock : Block` has k `:UnitBlock` and T `:NetworkBlock` recursively
- Problem data split between them: constraints only in `UCBlock`
- `Modification` sent to all `Solver` attached to `Block` and its ancestors
- `load()`-ed from text files or `[de]serialize()`-d to / from `netCDF` files

But `Solver` do not compose as easily as `Block`, sir!

- Composing "easy" problems most often makes a "much harder" one:
  integer MCF $\in \mathcal{P}$, integer MMCF $\in \mathcal{NP}$-hard

[27] Knueven, Ostrowski, Watson "On mixed-integer [...] formulations for the unit commitment problem" INFORMS JoC 2020
[28] Bacci, F., Gentile, Tavlaridis-Gyparakis "New MINLP Formulations for the Unit Commitment Problem [...]" *OR* 2024
[29] F., Gentile "Perspective cuts for a class of convex 0-1 mixed integer programs" *Math.Prog.* 2006

But `Solver` do not compose as easily as `Block`, sir!

- Composing "easy" problems most often makes a "much harder" one: single-thermal-UC $\in \mathcal{P}$, general UC $\in \mathcal{NP}$-hard

[27] Knueven, Ostrowski, Watson "On mixed-integer [...] formulations for the unit commitment problem" INFORMS JoC 2020

[28] Bacci, F., Gentile, Tavlaridis-Gyparakis "New MINLP Formulations for the Unit Commitment Problem [...]" *OR* 2024

[29] F., Gentile "Perspective cuts for a class of convex 0-1 mixed integer programs" *Math.Prog.* 2006

But `Solver` do not compose as easily as `Block`, sir!

- Composing "easy" problems most often makes a "much harder" one: single-thermal-UC $\in \mathcal{P}$, general UC $\in \mathcal{NP}$-hard
- Structure is brittle: specialised algorithms just stop working
  $\Longrightarrow$ must bump-up to more general algorithms
- More general algorithms don't "understand" specialised structure
  $\Longrightarrow$ must change problem "format" to one it "understands"
- "Abstract representation": any # of groups of `Variable` (any type) and (not necessarily algebraic) `Constraint`, one (any-valued) `Objective`
- Group = single, `vector`, `boost::multi_array` (of `list` for dynamic)

---

[27] Knueven, Ostrowski, Watson "On mixed-integer [...] formulations for the unit commitment problem" INFORMS JoC 2020

[28] Bacci, F., Gentile, Tavlaridis-Gyparakis "New MINLP Formulations for the Unit Commitment Problem [...]" *OR* 2024

[29] F., Gentile "Perspective cuts for a class of convex 0-1 mixed integer programs" *Math.Prog.* 2006

But `Solver` do not compose as easily as `Block`, sir!

- Composing "easy" problems most often makes a "much harder" one: single-thermal-UC $\in \mathcal{P}$, general UC $\in \mathcal{NP}$-hard
- Structure is brittle: specialised algorithms just stop working
  $\implies$ must bump-up to more general algorithms
- More general algorithms don't "understand" specialised structure
  $\implies$ must change problem "format" to one it "understands"
- "Abstract representation": any # of groups of `Variable` (any type) and (not necessarily algebraic) `Constraint`, one (any-valued) `Objective`
- Group = single, `vector`, `boost::multi_array` (of `list` for dynamic)
- A `Block` can have multiple different "abstract representations"; e.g., flow / knapsack formulations in MMCF

---

[27] Knueven, Ostrowski, Watson "On mixed-integer [...] formulations for the unit commitment problem" INFORMS JoC 2020

[28] Bacci, F., Gentile, Tavlaridis-Gyparakis "New MINLP Formulations for the Unit Commitment Problem [...]" *OR* 2024

[29] F., Gentile "Perspective cuts for a class of convex 0-1 mixed integer programs" *Math.Prog.* 2006

But `Solver` do not compose as easily as `Block`, sir!

- Composing "easy" problems most often makes a "much harder" one: single-thermal-UC $\in \mathcal{P}$, general UC $\in \mathcal{NP}$-hard
- Structure is brittle: specialised algorithms just stop working
  $\implies$ must bump-up to more general algorithms
- More general algorithms don't "understand" specialised structure
  $\implies$ must change problem "format" to one it "understands"
- "Abstract representation": any # of groups of `Variable` (any type) and (not necessarily algebraic) `Constraint`, one (any-valued) `Objective`
- Group = single, `vector`, `boost::multi_array` (of `list` for dynamic)
- A `Block` can have multiple different "abstract representations"; e.g., `ThermalUnitBlock` provides 3bin, T[27], "exact" DP-based, p_t, S-U, S-D[28]
- `Variable` & `Constraint` can be many $\implies$ dynamic representation; e.g.,

---

[27] Knueven, Ostrowski, Watson "On mixed-integer [...] formulations for the unit commitment problem" INFORMS JoC 2020
[28] Bacci, F., Gentile, Tavlaridis-Gyparakis "New MINLP Formulations for the Unit Commitment Problem [...]" *OR* 2024
[29] F., Gentile "Perspective cuts for a class of convex 0-1 mixed integer programs" *Math.Prog.* 2006

But `Solver` do not compose as easily as `Block`, sir!

- Composing "easy" problems most often makes a "much harder" one: single-thermal-UC $\in \mathcal{P}$, general UC $\in \mathcal{NP}$-hard
- Structure is brittle: specialised algorithms just stop working
  $\implies$ must bump-up to more general algorithms
- More general algorithms don't "understand" specialised structure
  $\implies$ must change problem "format" to one it "understands"
- "Abstract representation": any # of groups of `Variable` (any type) and (not necessarily algebraic) `Constraint`, one (any-valued) `Objective`
- Group = single, `vector`, `boost::multi_array` (of `list` for dynamic)
- A `Block` can have multiple different "abstract representations"; e.g., `ThermalUnitBlock` provides 3bin, T[27], "exact" DP-based, p_t, S-U, S-D[28]
- `Variable` & `Constraint` can be many $\implies$ dynamic representation; e.g.,

---

[27] Knueven, Ostrowski, Watson "On mixed-integer [...] formulations for the unit commitment problem" INFORMS JoC 2020
[28] Bacci, F., Gentile, Tavlaridis-Gyparakis "New MINLP Formulations for the Unit Commitment Problem [...]" *OR* 2024
[29] F., Gentile "Perspective cuts for a class of convex 0-1 mixed integer programs" *Math.Prog.* 2006

But `Solver` do not compose as easily as `Block`, sir!

- Composing "easy" problems most often makes a "much harder" one: single-thermal-UC $\in \mathcal{P}$, general UC $\in \mathcal{NP}$-hard
- Structure is brittle: specialised algorithms just stop working
  $\implies$ must bump-up to more general algorithms
- More general algorithms don't "understand" specialised structure
  $\implies$ must change problem "format" to one it "understands"
- "Abstract representation": any # of groups of `Variable` (any type) and (not necessarily algebraic) `Constraint`, one (any-valued) `Objective`
- Group = single, `vector`, `boost::multi_array` (of `list` for dynamic)
- A `Block` can have multiple different "abstract representations"; e.g., `ThermalUnitBlock` provides 3bin, T[27], "exact" DP-based, p_t, S-U, S-D[28]
- `Variable` & `Constraint` can be many $\implies$ dynamic representation; e.g., path-based formulation of [M]MCF

---

[27] Knueven, Ostrowski, Watson "On mixed-integer [...] formulations for the unit commitment problem" INFORMS JoC 2020

[28] Bacci, F., Gentile, Tavlaridis-Gyparakis "New MINLP Formulations for the Unit Commitment Problem [...]" *OR* 2024

[29] F., Gentile "Perspective cuts for a class of convex 0-1 mixed integer programs" *Math.Prog.* 2006

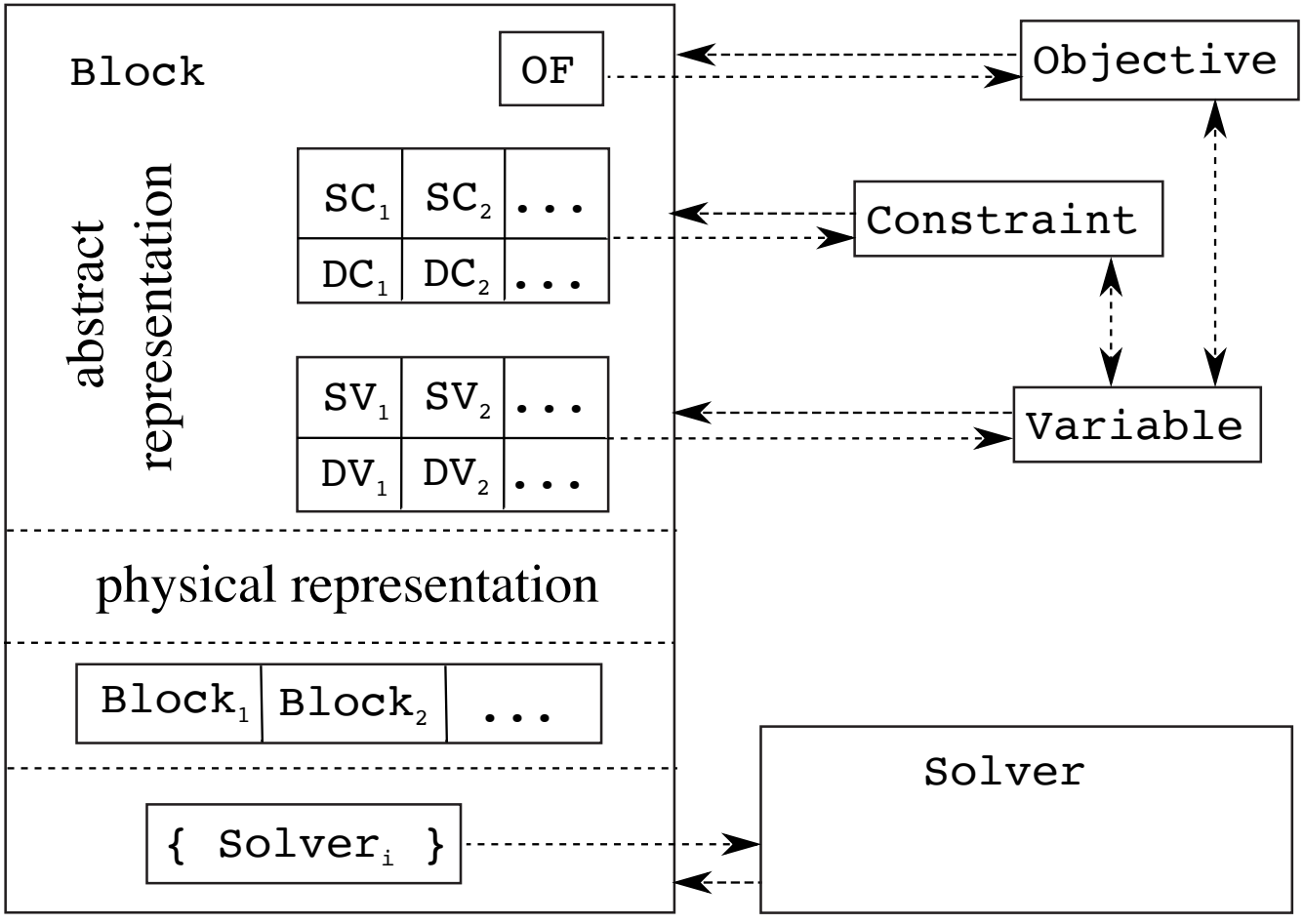
But `Solver` do not compose as easily as `Block`, sir!

- Composing "easy" problems most often makes a "much harder" one: single-thermal-UC $\in \mathcal{P}$, general UC $\in \mathcal{NP}$-hard
- Structure is brittle: specialised algorithms just stop working
  $\implies$ must bump-up to more general algorithms
- More general algorithms don't "understand" specialised structure
  $\implies$ must change problem "format" to one it "understands"
- "Abstract representation": any # of groups of `Variable` (any type) and (not necessarily algebraic) `Constraint`, one (any-valued) `Objective`
- Group = single, `vector`, `boost::multi_array` (of `list` for dynamic)
- A `Block` can have multiple different "abstract representations"; e.g., `ThermalUnitBlock` provides 3bin, T[27], "exact" DP-based, p_t, S-U, S-D[28]
- `Variable` & `Constraint` can be many $\implies$ dynamic representation; e.g., Perspective Cuts in `ThermalUnitBlock`[29]

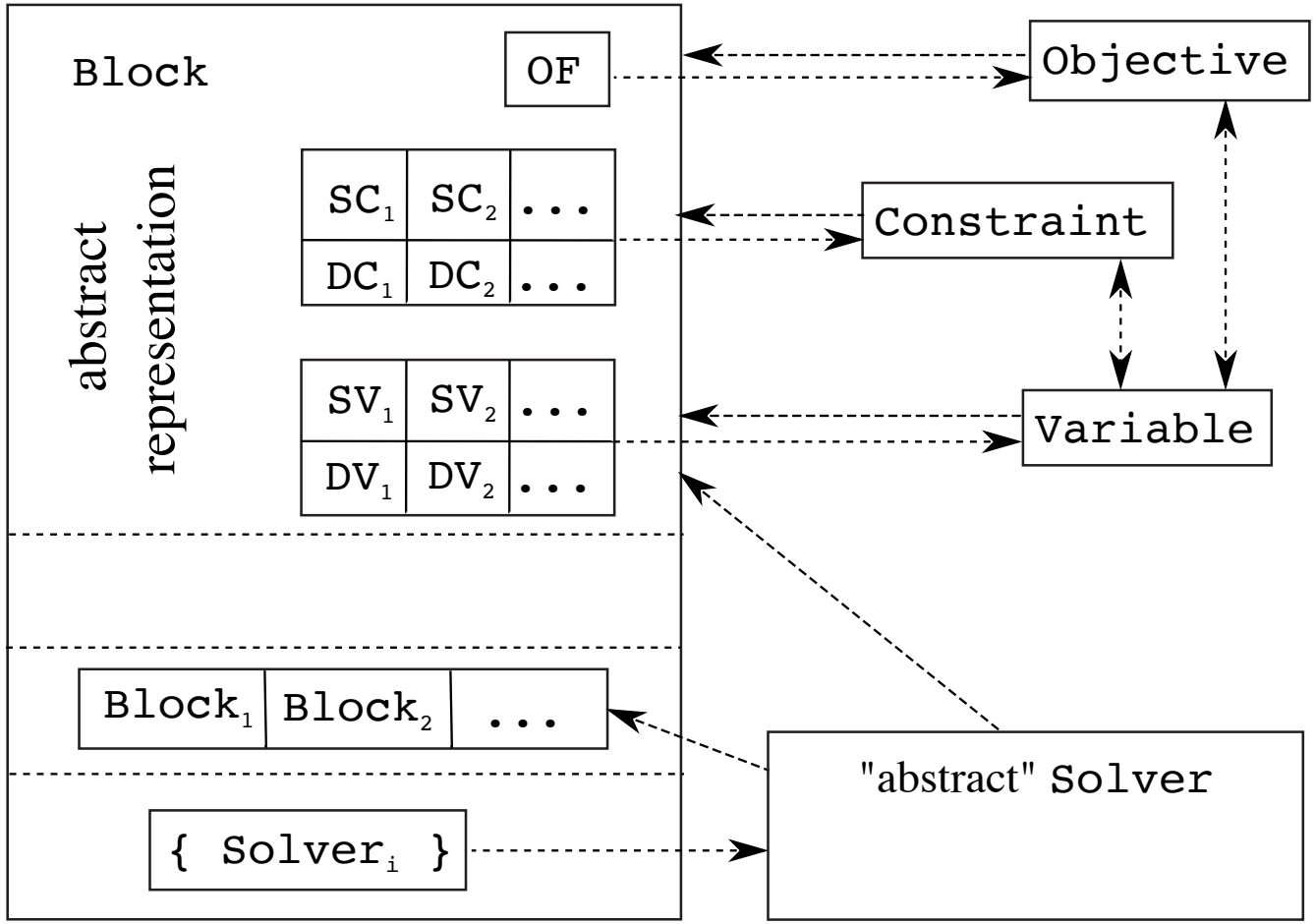
---

[27] Knueven, Ostrowski, Watson "On mixed-integer [...] formulations for the unit commitment problem" INFORMS JoC 2020
[28] Bacci, F., Gentile, Tavlaridis-Gyparakis "New MINLP Formulations for the Unit Commitment Problem [...]" *OR* 2024
[29] F., Gentile "Perspective cuts for a class of convex 0-1 mixed integer programs" *Math.Prog.* 2006

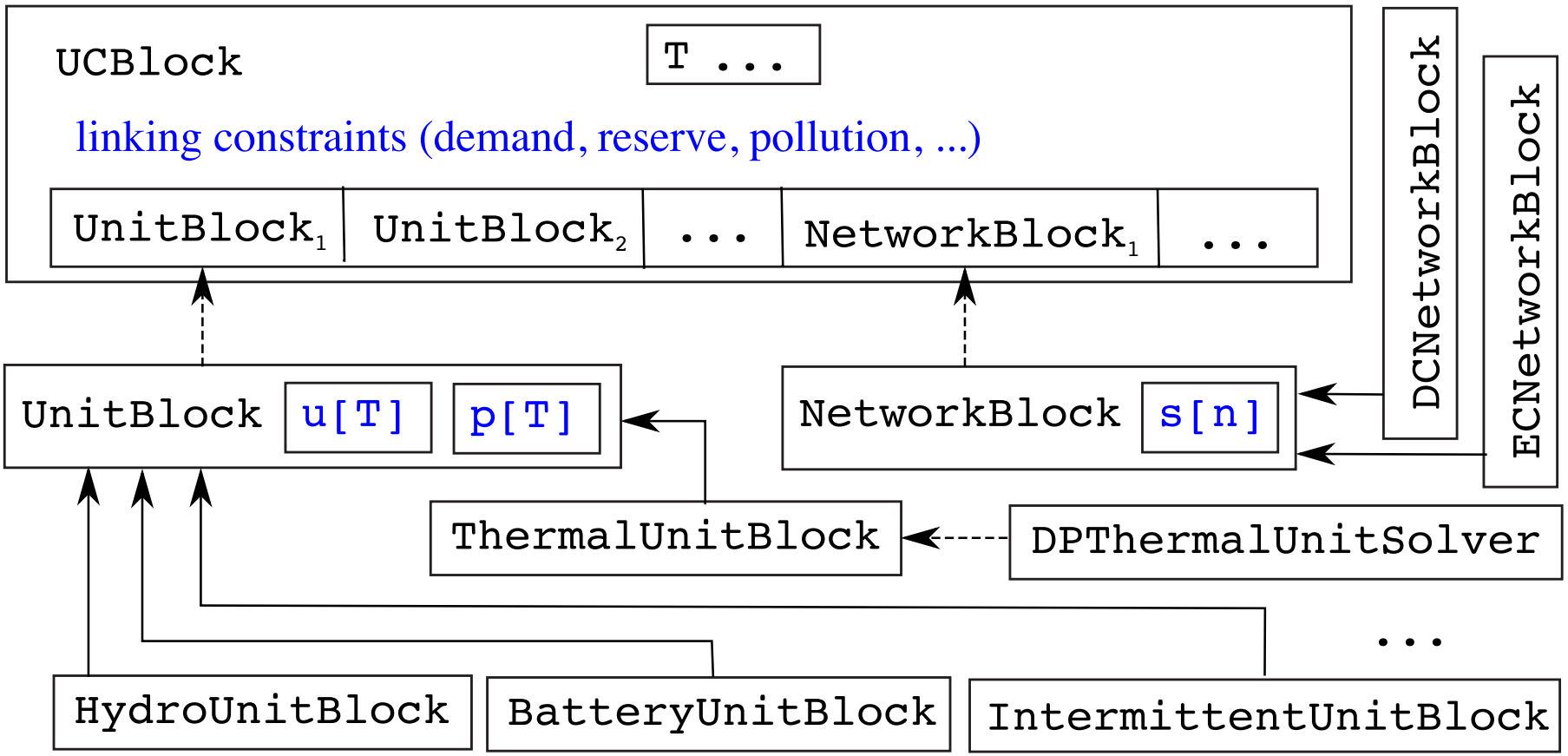
General-purpose `Solver` and "abstract representation"



- AR of sub-`Block` belongs to AR of the father `Block` (`Objective` sum)

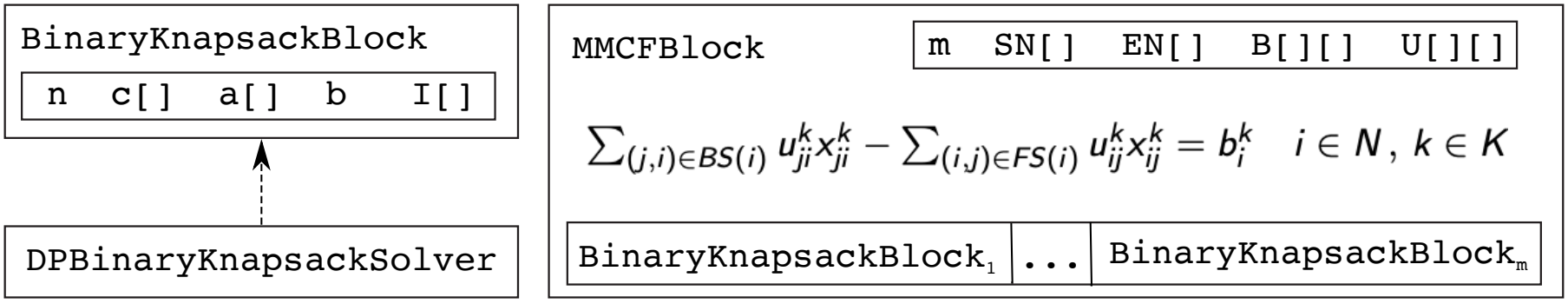


- AR of sub-`Block` belongs to AR of the father `Block` (`Objective` sum)
- "Abstract `Solver`" `generate_static_[variable/constraint]()` on-demand, piecemeal (selected groups of `Variable` / `Constraint`)
- Iteratively `generate_dynamic_[variable/constraint]()` using current (primal, and dual if $\exists$) solution



- Energy demand `Constraint` link `UCBlock` with its sub-`Block`
- Power / commitment `Variable` of each `UnitBlock` and injection `Variable` of each `NetworkBlock` link them with the `UCBlock`
- This always conceptually true even if `Constraint` / `Variable` not explicitly `generate_d`

There are always more ways to (re)formulate a problem



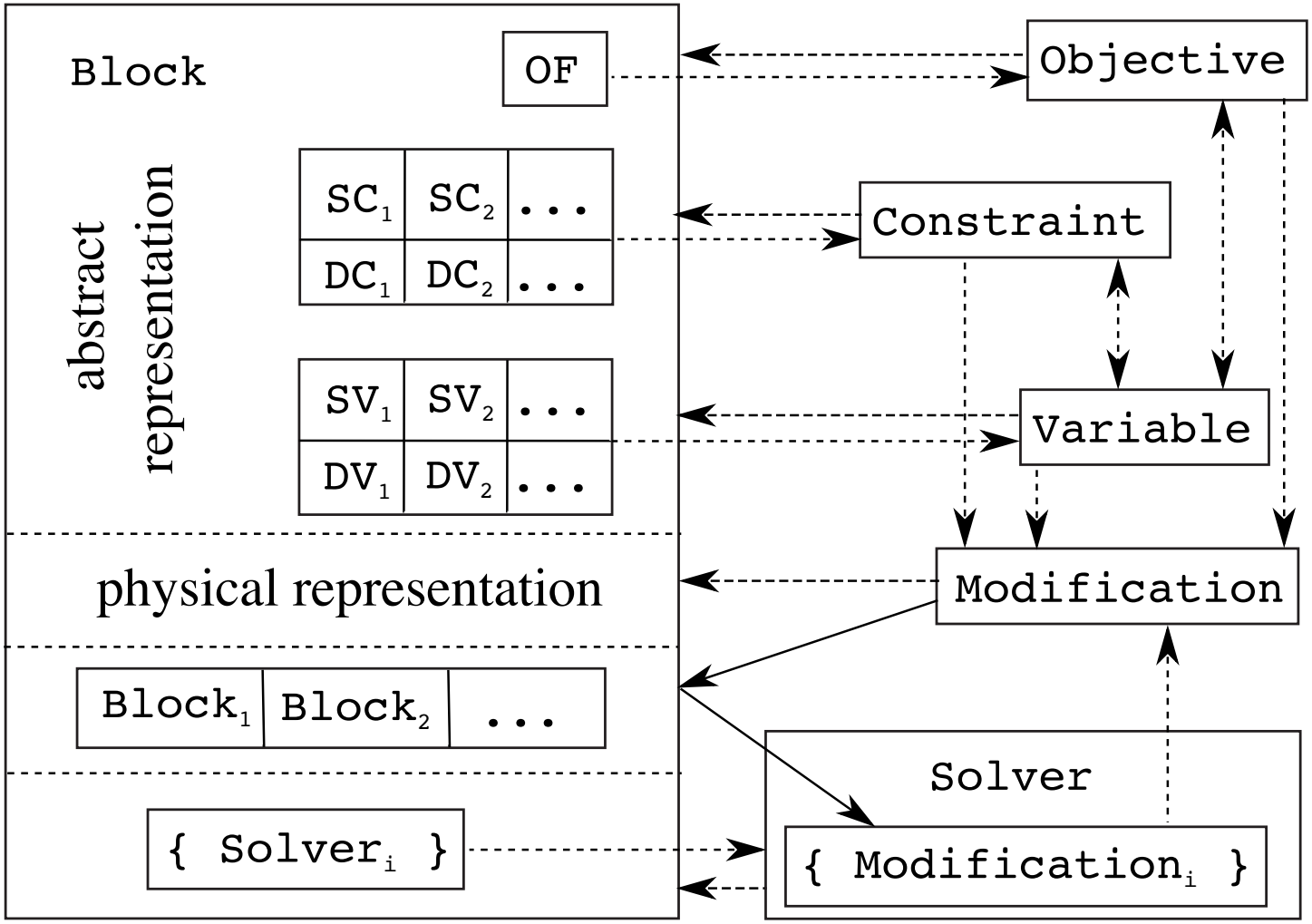
- A `Block` can have one of the many different formulations that are manually, painstakingly programmed into it (forget "auto" here)
- `[R/O/C]BlockConfig` allow to `Configure` all aspect of a `Block` and all its sub-`Block`, recursively (will see)
- Of course, appropriate sub-`Block` must have been manually, painstakingly programmed, typically together with appropriate specialised `Solver`
- This does not cater for multiple reformulations at the same time, but `R3Block` does (if manually, painstakingly programmed)

Multiple (re)formulations in flight: `R3Block`

- Often reformulation crucial, but also relaxation or restriction: `get_R3_Block()` produces one, possibly using sub-`Blocks`'
- Obvious special case: copy (clone) should always work
- Available `R3Block` `:Block`-specific, a `:Configuration` needed
- `R3Block` completely independent (new `Variable` / `Constraint` ...), useful for algorithmic purposes (branch, fix, solve, ...)
- Solution of `R3Block` useful to `Solver` for original `Block`: `map_back_solution()` (best effort in case of dynamic `Variable`)
- Sometimes keeping `R3Block` in sync with original necessary: `map_forward_Modification()`, task of original `Block`
- `map_forward_solution()` and `map_back_Modification()` useful, e.g., dynamic generation of `Variable`/`Constraint` in the `R3Block`
- `:Block` is in charge of all this, thus decides what it supports

- An "abstract" Solver may want to change the AR for algorithmic reasons
- :Block decides which AR changes it supports (throws exception if not)
- Block are "Janus", have "two faces" that must be kept in synk
- Nontrivial: the AR does not "know" the physical one
- Modification are issued to Block to be forwarded to interested Solver
- "Abstract Modification" issued by AR to Block opportunistically exploited to keep both representations in sync
- Specialized Solver disregard "abstract Modification", "abstract" Solver disregard standard ("physical") Modification
- concerns_Block() and anyone_there() to avoid useless Modification too coarse mechanisms, improvements needed ("types" of Modification)

Much less obvious than I initially envisioned



- Programming a `:Block` is nothing like writing an algebraic model: two representations, `Modification` ... (although `AbstractBlock` $\exists$)
- A lot of other minor details to be taken care of (will see)
- A quite unique structure and corresponding unique capabilities

Outline

Decomposition-aware modelling systems and solvers

SMS++ : design goals

SMS++ : basic ideas

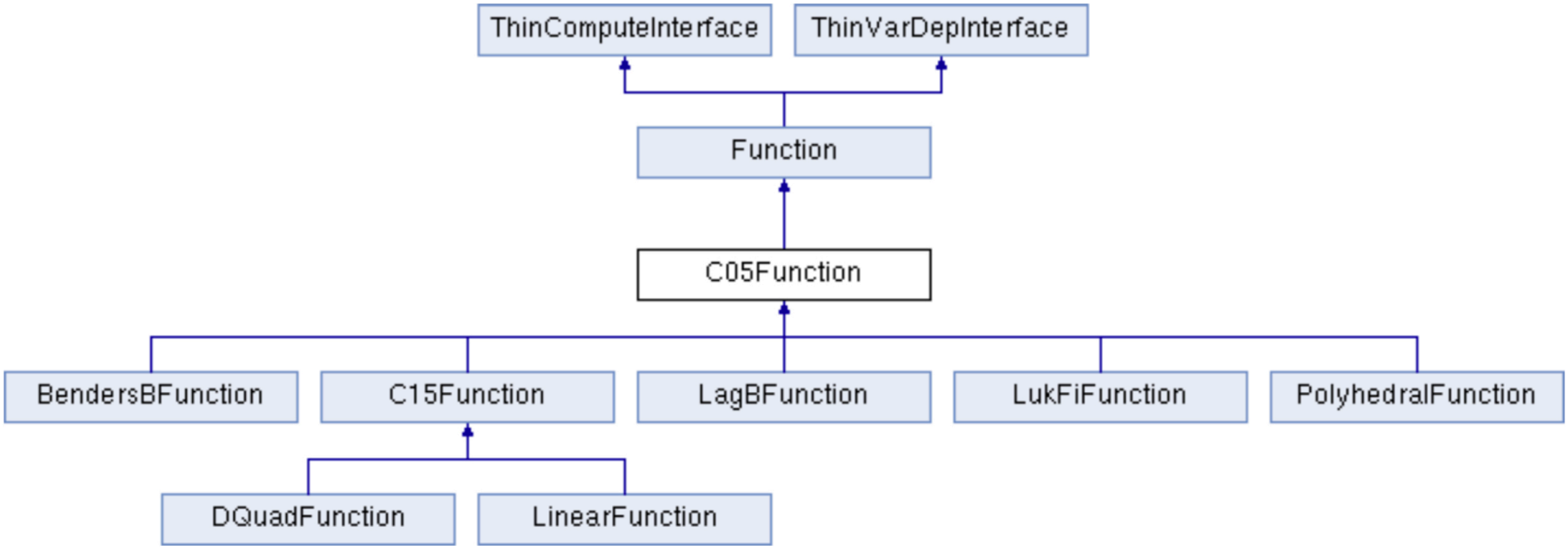
Many other necessary pieces

SMS++ : existing Block and Solver

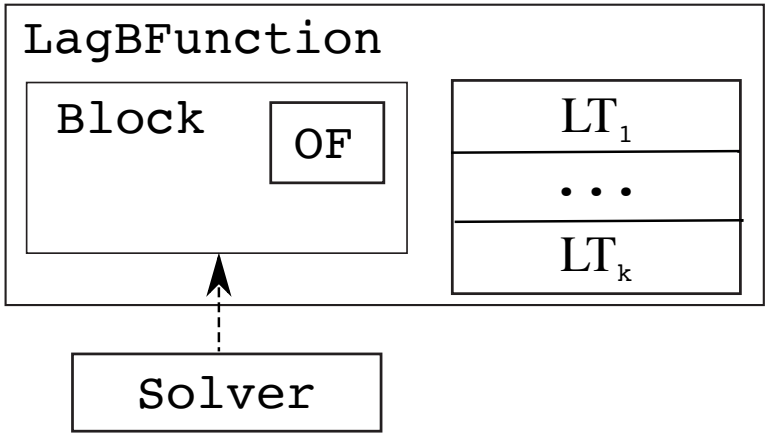
More advanced Block / Solver

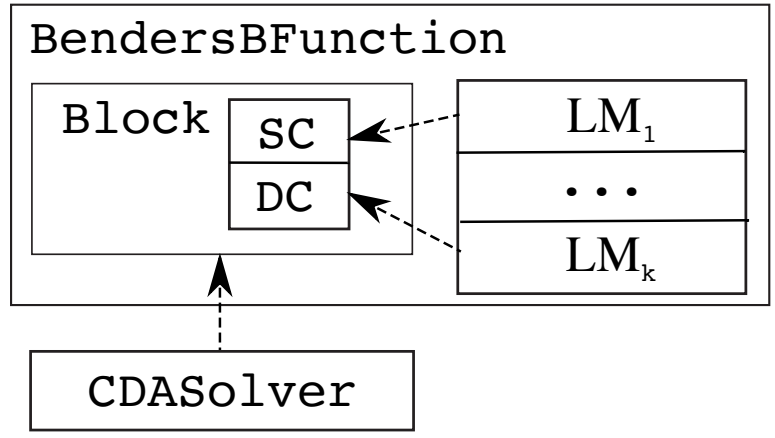
SMS++ and beyond: (some of) the missing pieces

Conclusions (for good)



- Real-valued `Function` of a set of `[Col]Variable` (a single $\mathbb{R}$ / $\mathbb{Z}$)
- Must be `compute()`-d w.r.t. the current value of the `[Col]Variable`, possibly a costly operation (`:ThinComputeInterface`)
- `C05Function` / `C15Function` have (not necessarily continuous) 1^{st} / 2^{nd} order information (vertical / diagonal linearizations)
- Local / global pools of linearizations
- "Easy" `Function` (linear, quadratic, polyhedral, ...) with no overhead





- `LagBFunction` $\equiv$ dual function $\varphi(\lambda) = \min\{ f(x) + (\lambda LT)x : x \in X \}$
  for (almost) any `Block` (B) $\min\{ f(x) : x \in X \}$
- `BendersBFunction` $\equiv$ value function
  $$v(y) = \min\{ f(x) : g(x) \leq LMy : x \in X \}$$
  for (almost) any `Block` (B) $\min\{ f(x) : g(x) \leq 0 , x \in X \}$
- Both are `:Block` and `:C05Function`, with (B) being the only sub-`Block`
- Use generic `[CDA]Solver` to `compute()` ($\approx$ just call its `compute()`)
- Store pools of primal / dual `Solution` corresponding to linearizations
- Any change in (B) is mapped in changes of F-values / the pools

- `Block` can be (write) `lock()`-ed to ensure atomic changes
- `lock()`-ing a `Block` automatically `lock()`s all inner `Block` (recursively)
- Analogously for `read_lock()`, any # of concurrent reads
- `lock()` (but not `read_lock()`) sets an owner and records its `std::thread::id`; other `lock()` from the same thread fail (`std::mutex` would not work there)
- Write starvation not handled yet
- `Solver`'s `compute()` must be thread-safe (`std::recursive_mutex`)
- `Solver` / `ThinComputeInterface` can be "lent ID" (solving a sub-`Block`)
- `Solver`'s `list<Modification>` under an "active guard" (`std::atomic`)
- General `State` of `Solver` for checkpointing (and reoptimization)

- `Block` produces `Solution` object, possibly using its sub-`Blocks`'
  $\Longrightarrow$ tree-structured complex objects (as their `Block`)
- `Solution` can `read()` its own `Block` and `write()` itself back:
  `Block`-specific rather than `Solver`-specific
- `Solution` may save a specific subsets of primal / dual information
  $\Longrightarrow$ a(nother) tree-shaped `Configuration` (will see)
- `[de]serialize()`-d to / from `netCDF` files
- Linear combination of `Solution` supported $\Longrightarrow$ "less general"
  `Solution` may (automatically) convert to "more general" ones
- `ColVariableSolution:Solution` uses the abstract representation of any `Block` that only have `ColVariables` to read / write the solution
- `RowConstraintSolution:Solution` same for dual information, (`RowConstraint`), `ColRowSolution` for both

- `Block` a tree-shaped complex object $\Longrightarrow$
  `Configuration` for them (and other stuff) tree-shaped complex object
- But also `SimpleConfiguration<T> : Configuration`
  (`T` an `int`, a `double`, a `std::pair<>`, a `std::vector<>` ...)
- `[C/O/R]BlockConfiguration : Configuration` set [recursively]:
  - which dynamic `Variable` / `Constraint` are generated, how
    (`Solver`, time limit, parameters ...)
  - which `Solution` is produced (what is saved)
  - the `ComputeConfiguration : Configuration` of any
    `Constraint` / `Objective` that needs one
  - a bunch of other `Block` parameters
- `[R]BlockSolverConfiguration : Configuration` set [recursively]
  which `Solver` are attached to the `Block` and their
  `ComputeConfiguration:Configuration`
- Can be `clear()`-ed for cleanup
- `load()`-ed from text files or `[de]serialize()`-d to / from `netCDF` files

Change (beta)

- New concept still not in `develop`, actively developed
- `Change` $\approx$ `Modification` + data: tells both what changes and how
- `deserialize()`-d to from `netCDF` files (not smart pointer)
  $\Longrightarrow$ distributed algorithms
- `:Change` need be defined for each `:Block`, typically just calling `chg_*()` methods in specialised interface
- Very nifty concept: undo-`Change` can be produced when `Change` `apply()`-ed to `Block`
- Not all `Change` can be undo-`Change`-d (may be too complex)
- Can be used for very general algorithmic schemes (B&X, ...)
- `AbstractChange` possible, still to be implemented

A lot of other support stuff

- Everything (`Block`, `Configuration`, `Solver`, `Change`, ...) has a factory: `std::string` (name of the class) $\mapsto$ object of that class
- A methods factory for changing the physical representation without knowing of which `:Block` it exactly is (standardised interface for changing methods)
- `AbstractBlock` for constructing a model a-la algebraic language, can be derived for "general `Block` + specific part", reads `.mps` and `.lp` files
- `PolyhedralFunction[Block]`, very useful for decomposition
- `AbstractPath` for indexing any `Constranit` / `Variable` in a `Block`
- `FakeSolver:Solver` stashes away all `Modification`, `UpdateSolver:Solver` immediately forwards / R^{3}Bs them
- ...

Outline

Decomposition-aware modelling systems and solvers

SMS++ : design goals

SMS++ : basic ideas

Many other necessary pieces

SMS++ : existing Block and Solver

More advanced Block / Solver

SMS++ and beyond: (some of) the missing pieces

Conclusions (for good)

Main Existing :Block

- MCFBlock / MMCFBlock: single / multicommodity flow
- BinaryKnapsackBlock (actually mixed-integer)
- CapacitatedFacilityLocationBlock
- UCBlock for UC, several UnitBlock and NetworkBlock for components
- LagBFunction and BendersBFunction (both :{C05Function,Block})
- StochasticBlock realizes scenarios (ScenarioGenerator) into any Block (using methods factory)
- TwoStageStochasticBlock represents two-stage stochastic programs
- SDDPBlock represents multi-stage stochastic programs suitable for Stochastic Dual Dynamic Programming (will see)
- Several others at different stages of development

- MCF[Lemon]Solver: wrapper to MCFClass[30] / LEMON[31] for MCFBlock
- DPBinaryKnapsackSolver: classical DP for BinaryKnapsackBlock
- ThermalUnitDPSolver for ThermalUnitBlock (state-of-the-art)
- MILPSolver: constructs matrix-based representation of any "MIQP" Block, then derived classes to interface with Cplex, SCIP, Gurobi, Highs
- [Parallel]BundleSolver : CDASolver: SMS++ -native version of[32] (still shares some code), several (but not all) state-of-the-art tricks
- SDDPSolver: wrapper for SDDP solver StOpt[33] using StochasticBlock, BendersBFunction and PolyhedralFunction
- SDDPGreedySolver: greedy forward simulator for SDDPBlock
- Several others at different stages of development

[30] https://github.com/frangio68/Min-Cost-Flow-Class
[31] https://lemon.cs.elte.hu/trac/lemon
[32] https://gitlab.com/frangio68/ndosolver_fioracle_project
[33] https://gitlab.com/stochastic-control/StOpt

Outline

Decomposition-aware modelling systems and solvers

SMS++ : design goals

SMS++ : basic ideas

Many other necessary pieces

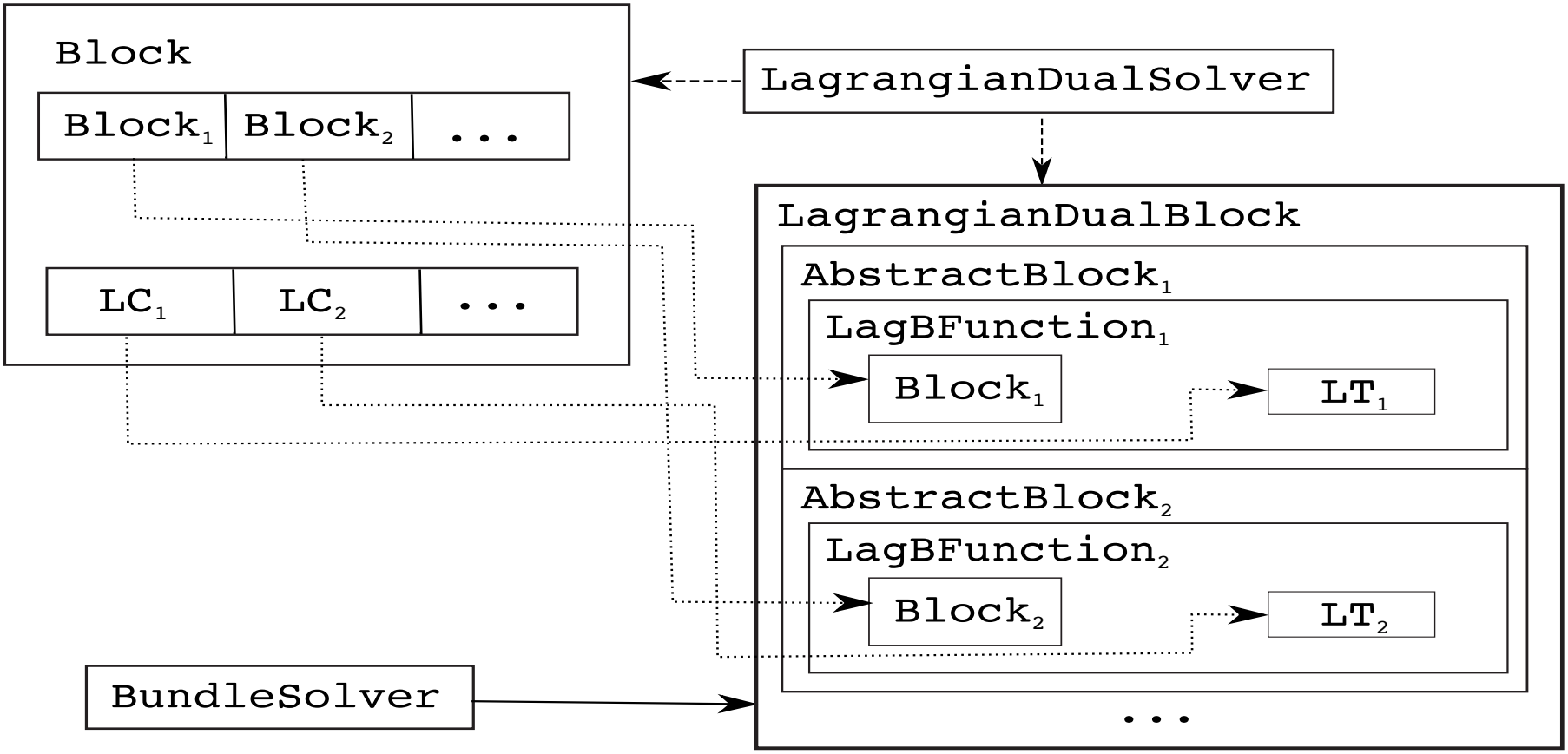
SMS++ : existing Block and Solver

More advanced Block / Solver

SMS++ and beyond: (some of) the missing pieces

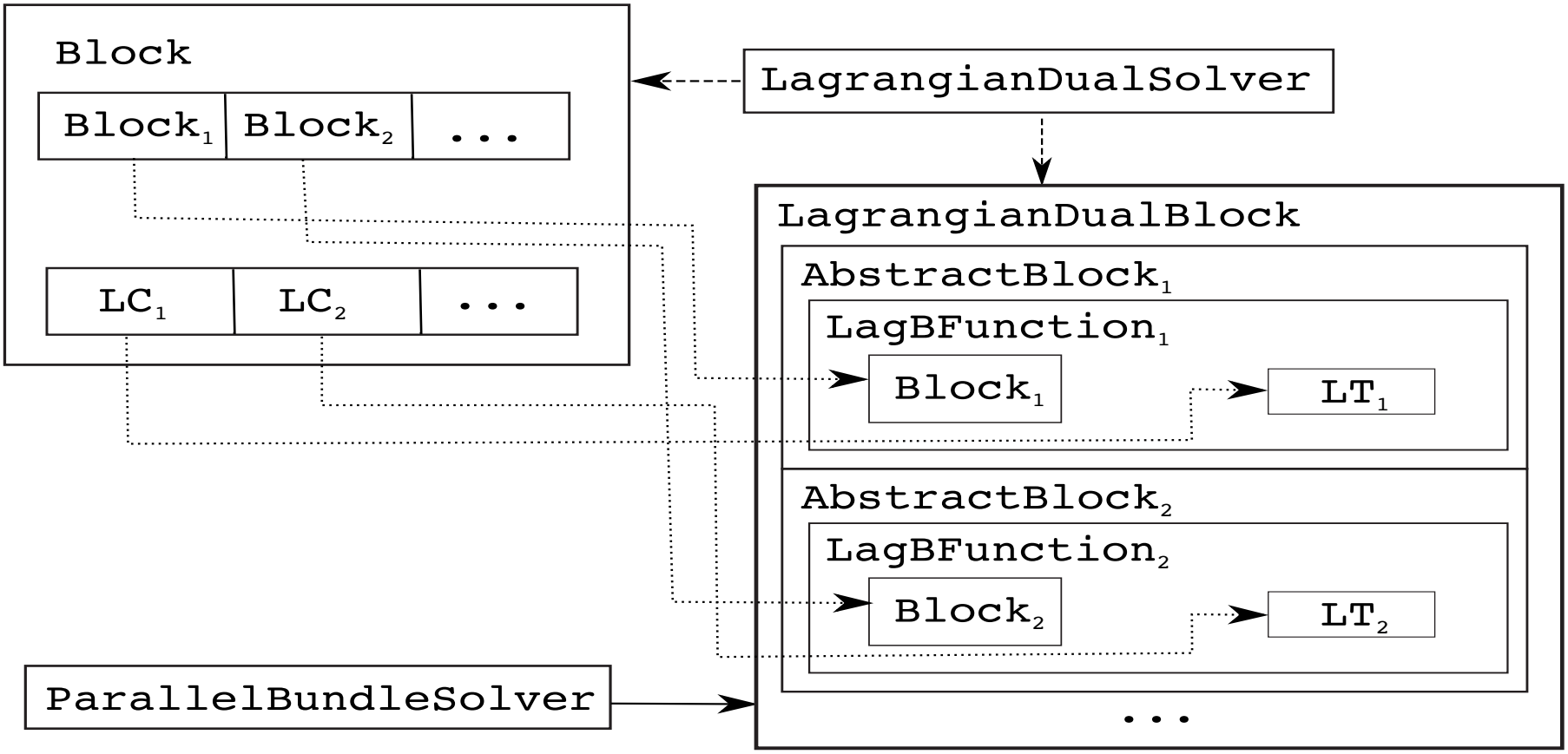
Conclusions (for good)

Our Masterpiece: LagrangianDualSolver



- Works for any `Block` with natural block-diagonal structure
- Using `LagBFunction` stealthily constructs `LagrangianDualBlock` w.r.t. linking `Constraint`, R^{3}B-ing or "stealing" the inner `Block`
- Solves the Lagrangian Dual with appropriate `:CDASolver`

Our Masterpiece: LagrangianDualSolver



- Works for any `Block` with natural block-diagonal structure
- Using `LagBFunction` stealthily constructs `LagrangianDualBlock` w.r.t. linking `Constraint`, R³B-ing or "stealing" the inner `Block`
- Solves the Lagrangian Dual with appropriate `:CDASolver` possibly in parallel, provides dual and "convexified" solution in original `Block`

- Major improvement: weeks of work in days / hours if `Block` has the right form
- Multilevel nested parallel heterogeneous decomposition by design
  (when we'll have ironed out all the bugs, nontrivial work)
- Hopefully soon `BendersDecompositionSolver` (`BendersBFunction` $\exists$)

[34] Daniildis, Lemaréchal "On a primal-proximal heuristic in discrete optimization *Math.Prog.* 2005"
[35] F., Pan, Traversi, Wolfler Calvo "A Constraints-Aware Reweighted Feasibility Pump Approach" *O.R.Lett.* 2021
[36] Cacciola, Forel, F., Lodi "The Differentiable Feasibility Pump" *IPCO* 2025
[37] Dubost, Gonzalez, Lemaréchal "A primal-proximal heuristic applied to French Unitcommitment problem" *Math.Prog.* 2005"

- Major improvement: weeks of work in days / hours if `Block` has the right form
- Multilevel nested parallel heterogeneous decomposition by design
  (when we'll have ironed out all the bugs, nontrivial work)
- Hopefully soon `BendersDecompositionSolver` (`BendersBFunction` $\exists$)
- Yet, only provides lower bounds / convexified solutions

[34] Daniildis, Lemaréchal "On a primal-proximal heuristic in discrete optimization *Math.Prog.* 2005"
[35] F., Pan, Traversi, Wolfler Calvo "A Constraints-Aware Reweighted Feasibility Pump Approach" *O.R.Lett.* 2021
[36] Cacciola, Forel, F., Lodi "The Differentiable Feasibility Pump" *IPCO* 2025
[37] Dubost, Gonzalez, Lemaréchal "A primal-proximal heuristic applied to French Unitcommitment problem" *Math.Prog.* 2005"

- Major improvement: weeks of work in days / hours if `Block` has the right form
- Multilevel nested parallel heterogeneous decomposition by design
  (when we'll have ironed out all the bugs, nontrivial work)
- Hopefully soon `BendersDecompositionSolver` (`BendersBFunction` $\exists$)
- Yet, only provides lower bounds / convexified solutions
- New development: `PrimalProximalHeur` $\equiv$ the same-named heuristic[34]
- Rough sketch: compute convexified solution $\bar{x} \in conv(X)$, round $\tilde{x} = \lceil \bar{x} \rfloor$, add penalty term $M \| x - \tilde{x} \|$ in the objective ($x \in \{0, 1\}^n \implies$ linear)
- Issue: right value of M (a heuristic, anyway)
- Similar to feasibility pump[35,36] but built for the Lagrangian setting
- Quite good results on UC[37], just put it in the `BlockSolverConfig` and try

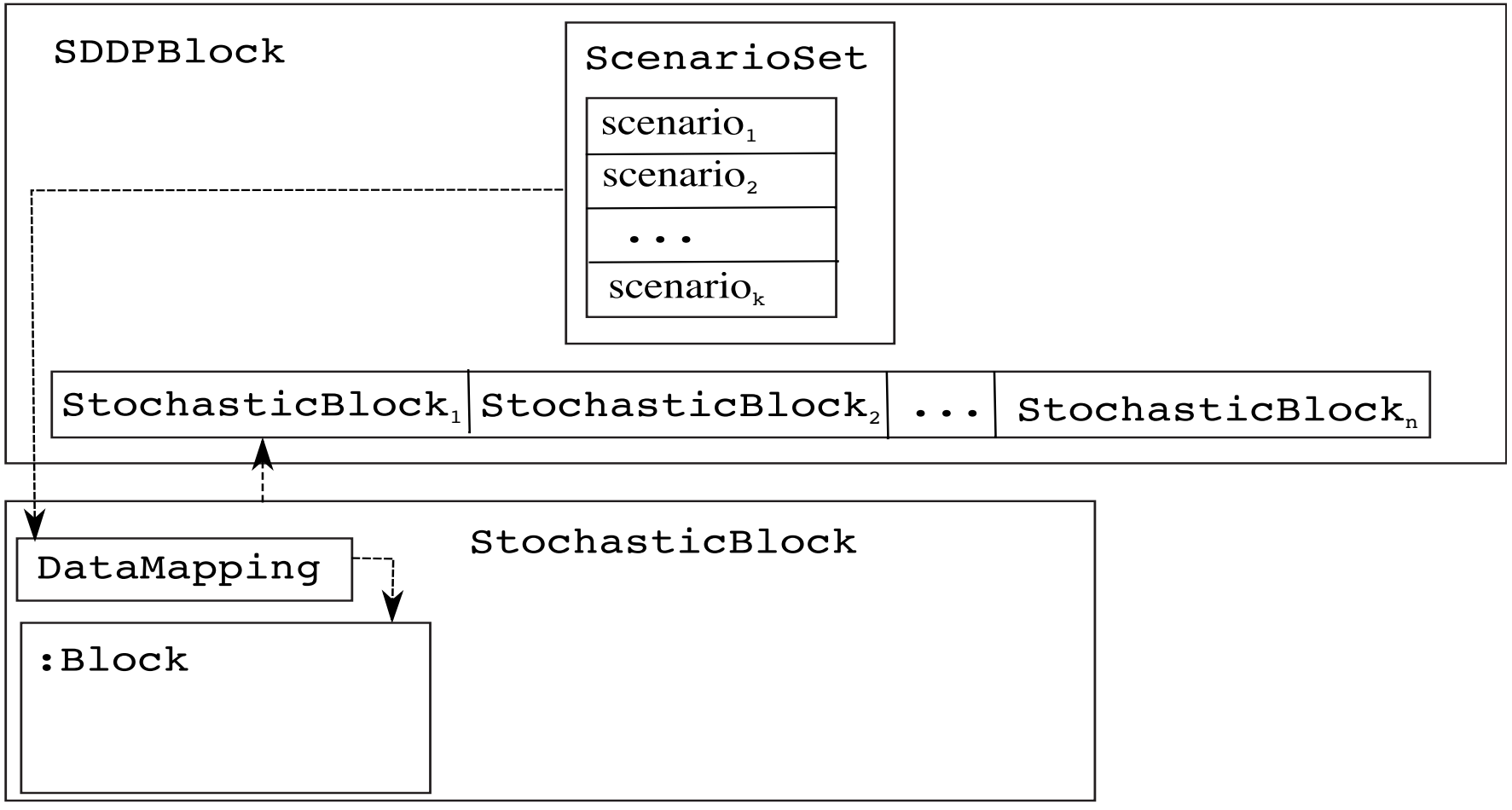
[34] Daniildis, Lemaréchal "On a primal-proximal heuristic in discrete optimization *Math.Prog.* 2005"
[35] F., Pan, Traversi, Wolfler Calvo "A Constraints-Aware Reweighted Feasibility Pump Approach" *O.R.Lett.* 2021
[36] Cacciola, Forel, F., Lodi "The Differentiable Feasibility Pump" *IPCO* 2025
[37] Dubost, Gonzalez, Lemaréchal "A primal-proximal heuristic applied to French Unitcommitment problem" *Math.Prog.* 2005"

$$\begin{aligned} \min\, & c^0x^0 + \textstyle\sum_{s\in S^1} \pi_s c_s^1 x_s^1 + \sum_{s\in S^2} \pi_s c_s^2 x_s^2 \\ & D^0x^0 \le b^0 \\ & E_s^0x^0 + E_s^1x_s^1 \le b_s^1 && s \in S^1 \\ & E_s^1x_s^1 + E_s^2x_s^2 \le b_s^2 && s \in S^2 \end{aligned}$$

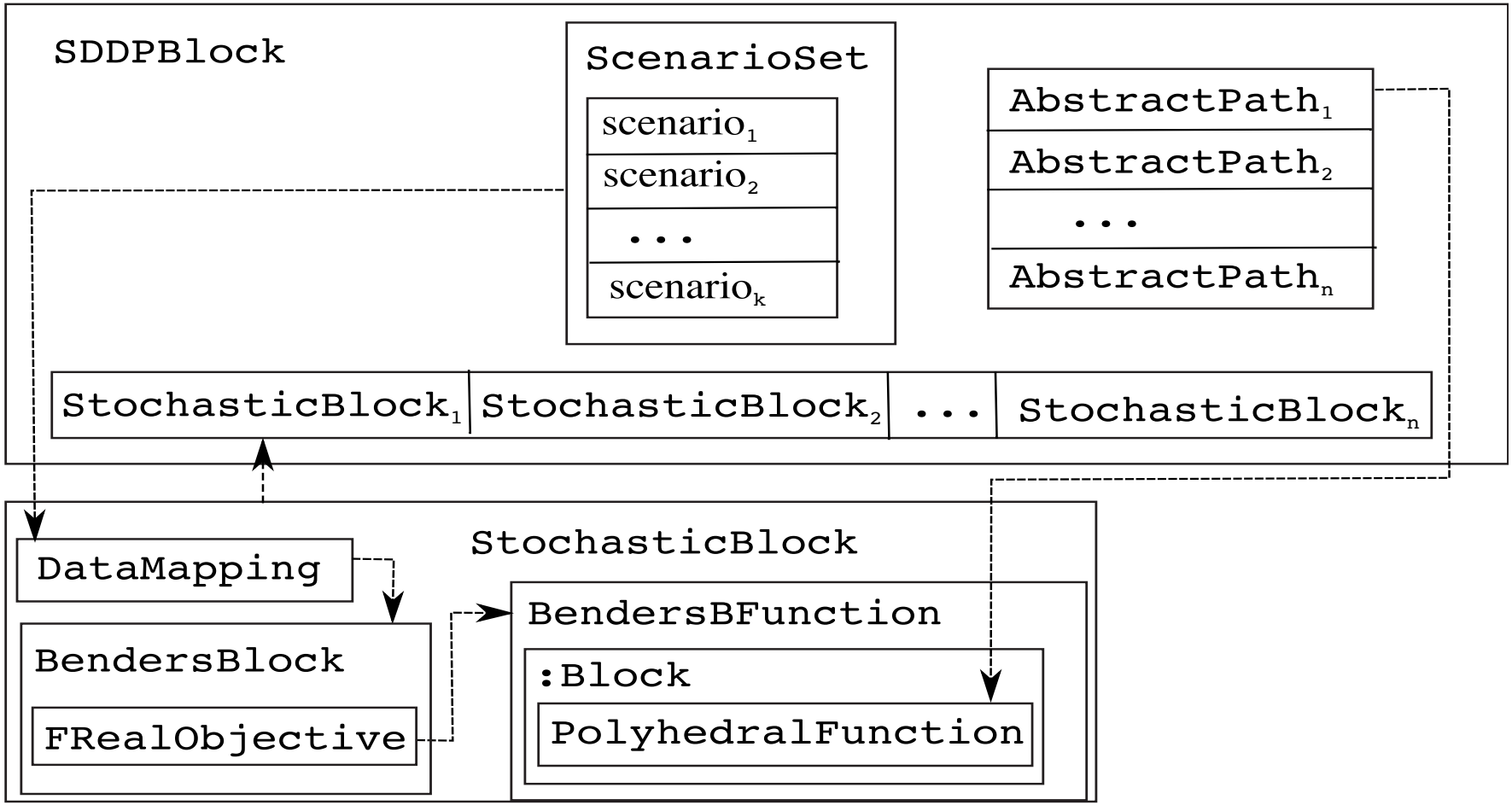
- Stagewise independent k-stage stochastic problem ($s \in S^{i+1}$ does not depend on $s \in S^i$) with consecutive couplings (constraints only link x^i with x^{i+1})
- $\equiv \min\{\, c^0x^0 + v^0(\,x^0\,) \,:\, D^0x^0 \le b^0 \,\}$ with recourse cost functions

  $v^0(\,x^0\,) = \sum_{s\in S^1} \min\{\, \pi_s c_s^1 x_s^1 + v^1(\,x_s^1\,) \,:\, E_s^1x_s^1 \le b_s^1 - E_s^0x^0 \,\}$

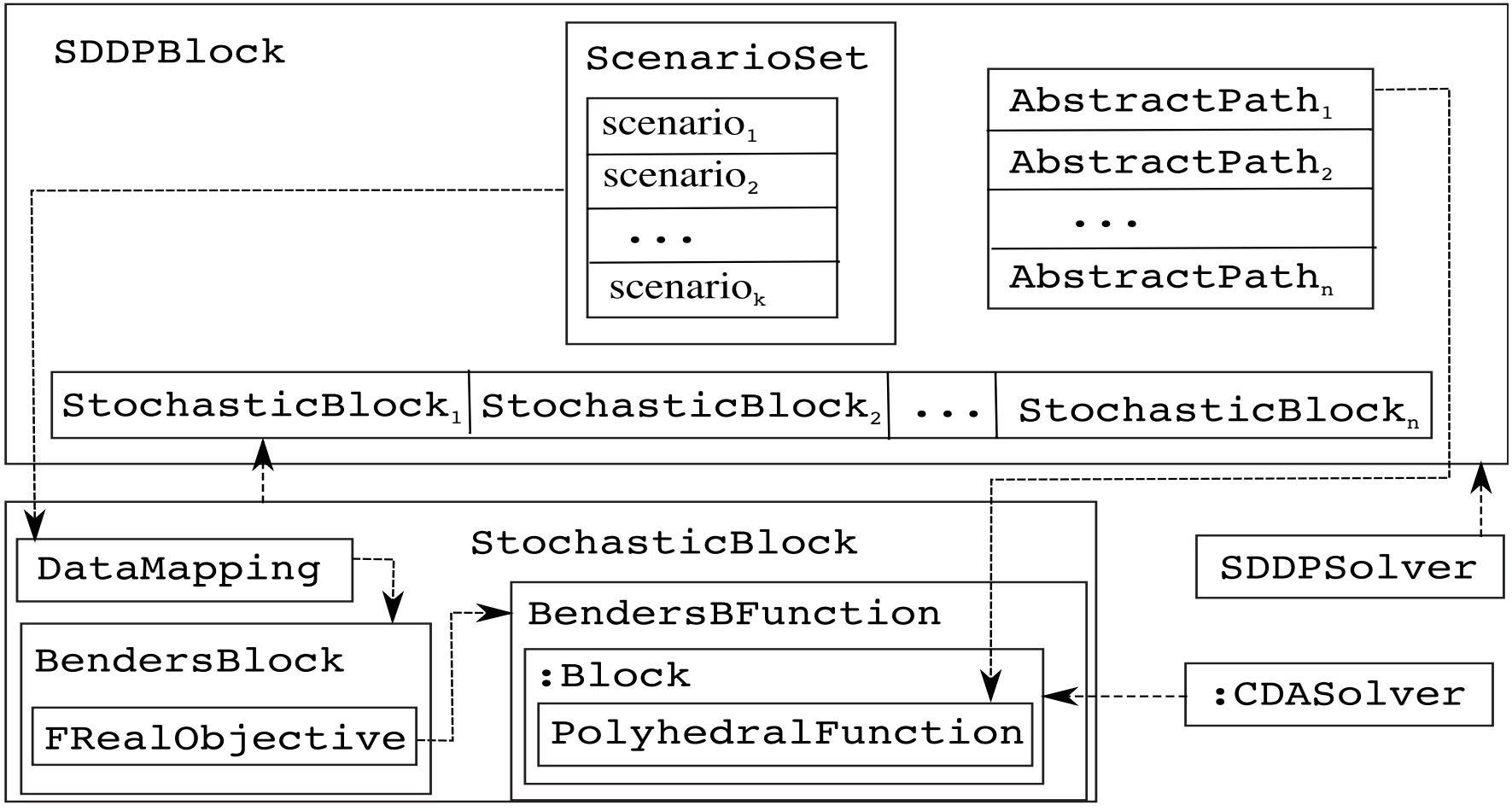
  $v^1(\,x^1\,) = \sum_{s\in S^2} \min\{\, \pi_s c_s^2 x_s^2 \,:\, E_s^2x_s^2 \le b_s^2 - E_s^1x^1 \,\}$
- Nested Benders': define approximations $\hat{v}^i(\cdot)$, then iteratively
  - forward pass: fixed x^{i-1}, solve stage i for primal / dual solutions x_s^i, w_s^i
  - backward pass: use w_s^i to generate Benders' cuts to improve $\hat{v}^{i-1}(\cdot)$
- At each stage different scenarios can be processed in parallel



- One `StochasticBlock` (arbitrary `Block`) per stage, `ScenarioGenerator` maps arbitrary scenario values (duals) into the `Block` (methods factory)



- One `StochasticBlock` (arbitrary `Block`) per stage, `ScenarioGenerator` maps arbitrary scenario values (duals) into the `Block` (methods factory)
- Benders' cuts from `BendersBFunction` $\rightarrow$ `PolyhedralFunction` in `Block`



- One StochasticBlock (arbitrary Block) per stage, ScenarioGenerator maps arbitrary scenario values (duals) into the Block (methods factory)
- Benders' cuts from BendersBFunction $\rightarrow$ PolyhedralFunction in Block
- Any CDASolver in BendersBFunction + SDDPSolver (wrapper for StOpt[33])

- Set N of generation / distribution units, κ_i identical copies of each $i \in N$
- For $\kappa = [\kappa_i]_{i \in N}$, investment cost $F(\kappa)$ ("easy") and operational cost

$$\begin{aligned} O(\kappa) = \min \ & \textstyle\sum_{i \in N} \sum_{j=1}^{\kappa_i} c_i(x_{i,j}) \\ \text{s.t. } & x_{i,j} \in X_i && j = 1, \ldots, \kappa_i \ , \ i \in N \\ & \textstyle\sum_{i \in N} \sum_{j=1}^{\kappa_i} A_i x_{i,j} \geq d \end{aligned}$$

- Everything convex $\implies$ all $i \in N$ produce identically at optimality $\implies$

$$\begin{aligned} O(\kappa) = \min \ & \textstyle\sum_{i \in N} \kappa_i c_i(x_i) \\ \text{s.t. } & x_i \in X_i && i \in N \\ & \textstyle\sum_{i \in N} \kappa_i A_i x_i \geq d \end{aligned}$$

- Extends to stochastic setting (S = scenarios, $\mathcal{N}$ = nonanticipativity)

$$\begin{aligned} O(\kappa) = \min \ & \textstyle\sum_{i \in N} \kappa_i \sum_{s \in S} \pi^s c_i^s(x_i^s) \\ \text{s.t. } & x_i^s \in X_i^s && i \in N \ , \ s \in S \\ & \textstyle\sum_{i \in N} \kappa_i \sum_{s \in S} A_i^s x_i^s \geq d \ , \ x \in \mathcal{N} \end{aligned}$$

A triply clever trick

- Investment problem $\min\{ F(\kappa) + O(\kappa) : \kappa \in K \}$:
  extremely hard since convexity assumption untrue

- Lagrangian relaxation triply clever:
  $\phi(\lambda, \kappa) = \lambda d + \sum_{i \in N} \kappa_i \min\{ c_i(x_i) - \lambda A_i x_i : x_i \in X_i \}$
  - decomposes into (many, easier, smaller) independent subproblems
  - automatically convexifies c and X
  - $\phi(\lambda, \kappa)$ is concave in λ and affine in κ

- Convexified version: $\underline{O}(\kappa) = \max\{ \phi(\lambda, \kappa) : \lambda \geq 0 \} = \phi(\lambda^*(\kappa), \kappa)$

- Convexified Investment Problem: $\min\{ F(\kappa) + \underline{O}(\kappa) : \kappa \in K \}$
  possibly the best trade-off between computability and accuracy

- Crucial: $[\, c_i(x_i^*(\lambda^*(\kappa))) - \lambda^*(\kappa) A_i x_i^*(\lambda^*(\kappa)) \,]_{i \in N} \in \partial \underline{O}(\kappa)$[38]
  $\Longrightarrow$ can use (inexact) bundle methods[39] or stabilised Benders' ones[40]

---

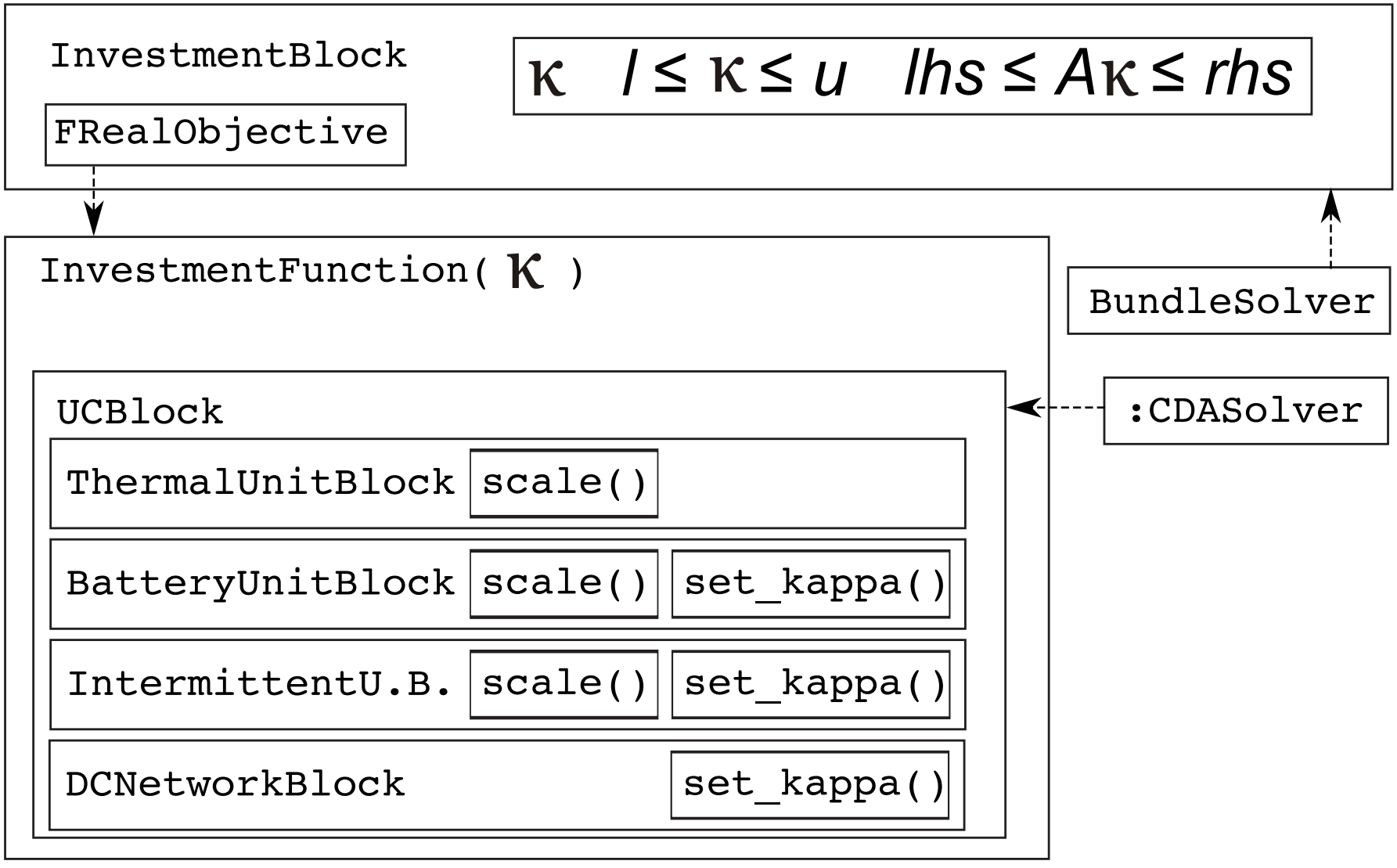
[38] van Ackooij, Oudjane "On supply and network investment in power systems" *4OR* 2024

[39] van Ackooij, F. "Incremental Bundle Methods Using Upper Models" *SIOPT* 2018

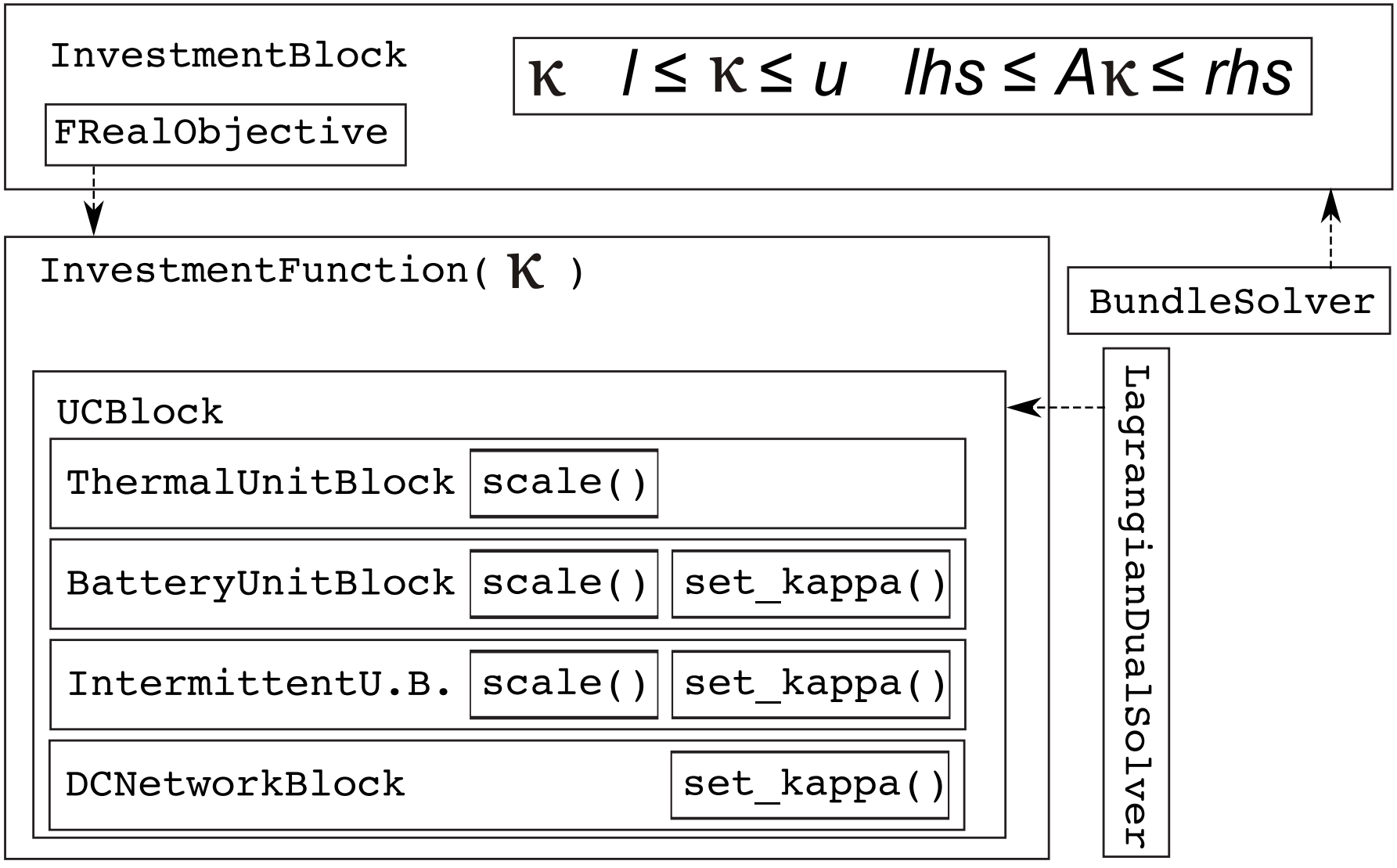
[40] van Ackooij, F., de Oliveira "Inexact Stabilized Benders' Decomposition Approaches [...]" *COAP* 2016

- Specific component for the Investment Problem
- Mostly relays on specialised `InvestmentFunction`: both a `C05Function` and a `Block` (sounds familiar?), computing $\underline{O}(\kappa)$
- Two kinds of continuous resources (κ_i):
  - upper / lower bounds on variables (network / unit capacities)
  - number of copies of existing `:UnitBlock`
- Requires support from `UCBlock` and the `:UnitBlock` / `:NetworkBlock`:
  - `scale()` in `ThermalUnitBlock`, `BatteryUnitBlock`, `IntermittentUnitBlock` (scales the whole unit)
  - `set_kappa()` in `BatteryUnitBlock`, `IntermittentUnitBlock`, `DCNetworkBlock` (only scales some RHSs)

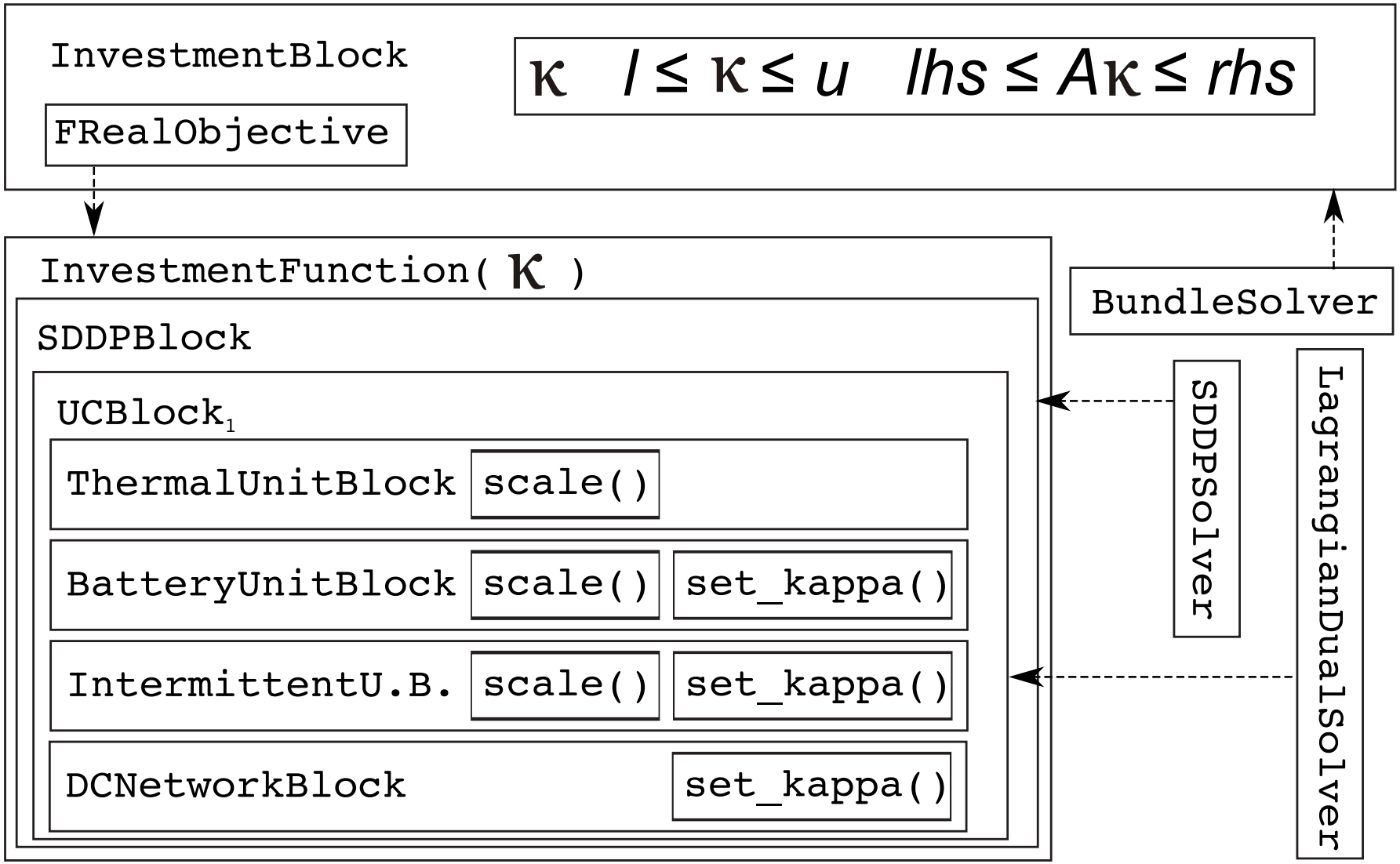
  then `UCBlock` has to scale the linking constraints when κ_i change
- Supports `SDDPBlock` with `UCBlock` inside (same changes to all stages)
- Can use any `CDASolver` (`LagrangianDualSolver`, `:MILPSolver`)



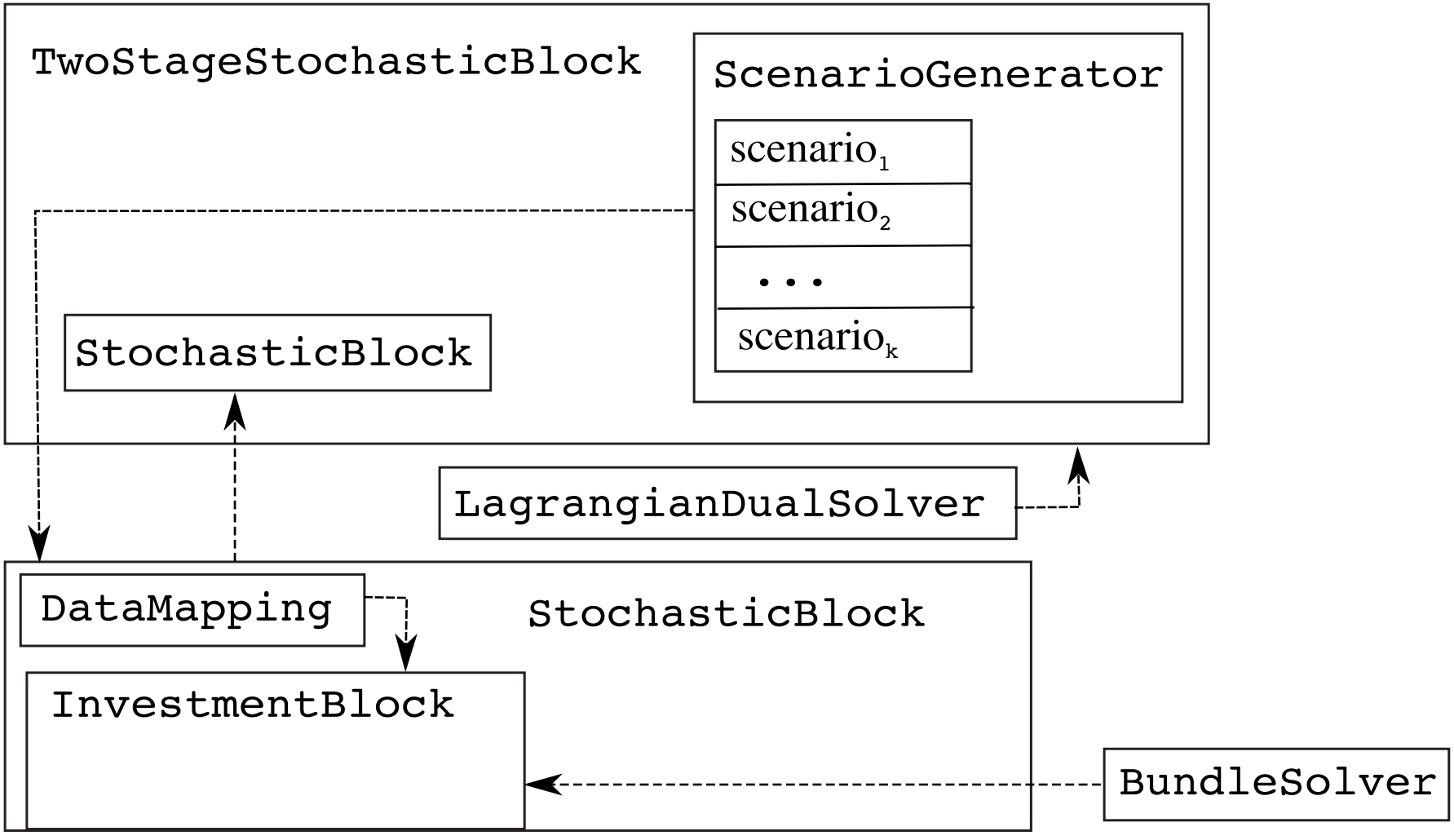
- Any :CDASolver $\Longrightarrow$



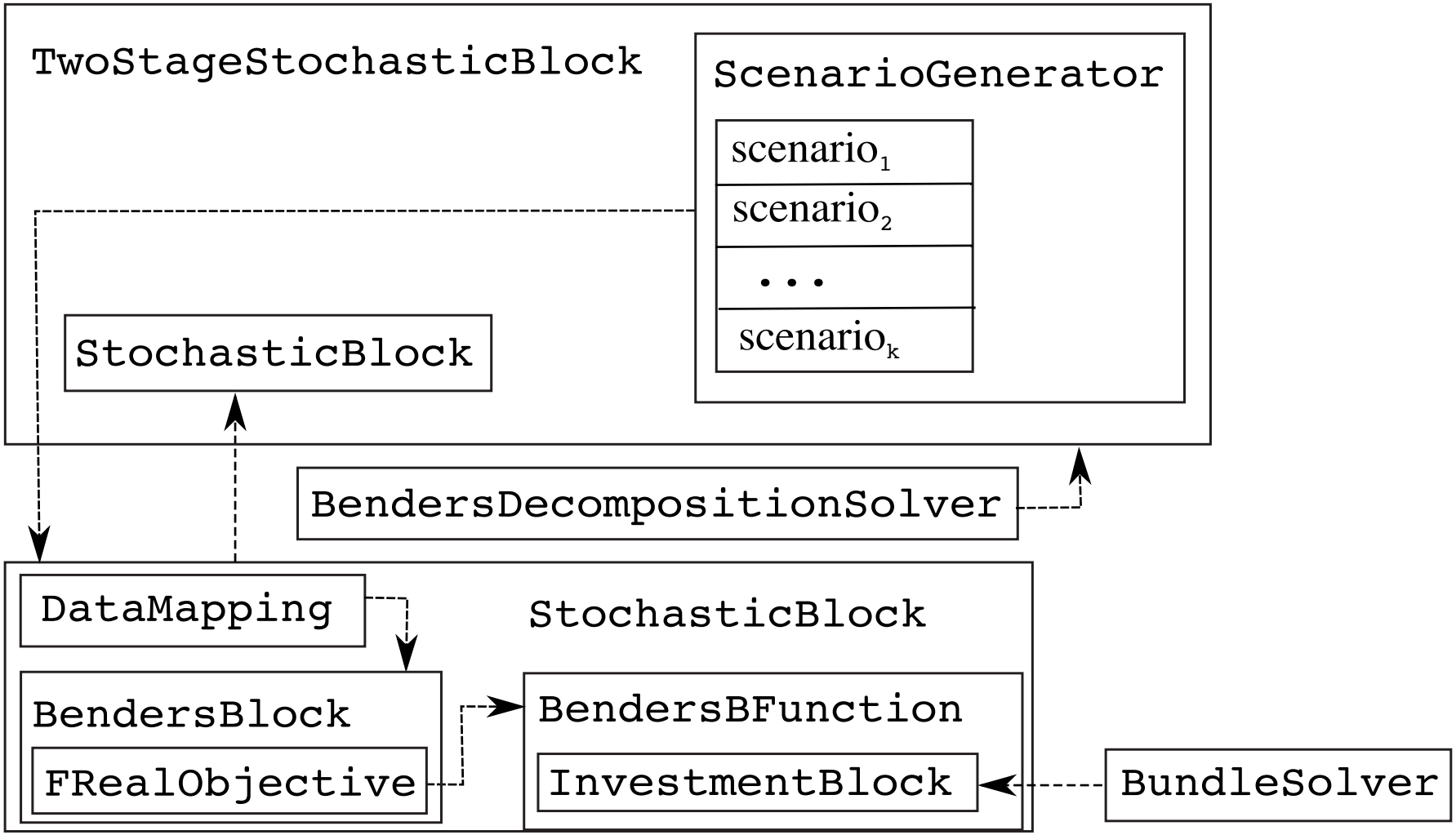
- Benders' with Lagrange inside: two-level decomposition



- Benders' with SDDP with Lagrange inside: three-level decomposition
- Scaling a `:Block` a general concept, may be upcasted to base `Block`



- Replicate linking variables, Lagrange-relax them



- Apply Benders' (but BendersDecompositionSolver yet to come)
- Lagrange inside SDDP inside Benders inside Lagrange / Benders ...
- Not your usual problem, but if we can solve this, we can solve yours

Outline

Decomposition-aware modelling systems and solvers

SMS++ : design goals

SMS++ : basic ideas

Many other necessary pieces

SMS++ : existing Block and Solver

More advanced Block / Solver

SMS++ and beyond: (some of) the missing pieces

Conclusions (for good)

The many things that we do not have (yet)

- A relaxation-agnostic Branch-and-X `Solver` (Filippo to the rescue!)
- Many other forms of (among many other things):
  - `Variable` (`Vector/MatrixVariable`, `FunctionVariable`, ...)
  - `Constraint` (`SOCConstraint`, `SDPConstraint`, `PDEConstraint`, `BilevelConstraint`, `EquilibriumConstraint`, ...)
  - `Objective` (`RealVectorObjective`, ...)
  - `Function` (`AlgebraicFunction`, ...)
- Better handling of many things (groups of stuff, `Modification`, ...)
- Interfaces with many other general-purpose solvers (`Couenne`, `OSISolverInterface`, OR-tools `CP-SAT Solver`, ...)
- Many many many more `:Block` and their specialised `:Solver`
- Translation layers from real modelling languages (AMPL, JuMP, ...)
- In a word: users / mindshare – chicken-and-egg problem

Catching the chicken (or is it the egg?)

- Chasing the large user base of `PyPsa`[41] via RESILIENT[42]

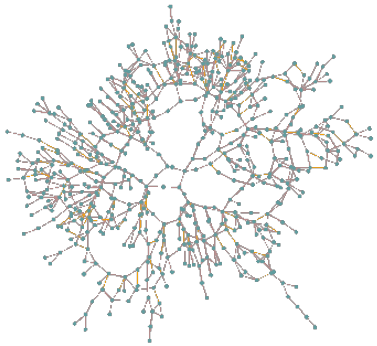
[41] https://pypsa.org

[42] https://resilient-project.github.io

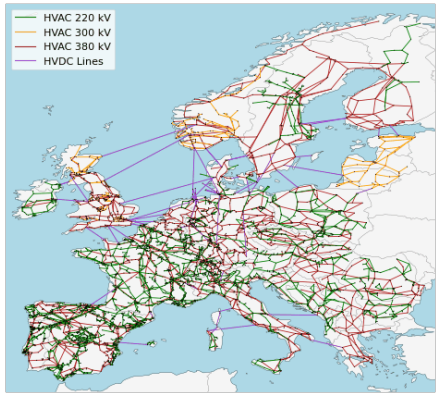
Catching the chicken (or is it the egg?)

- Chasing the large user base of `PyPsa`[41] via RESILIENT[42]

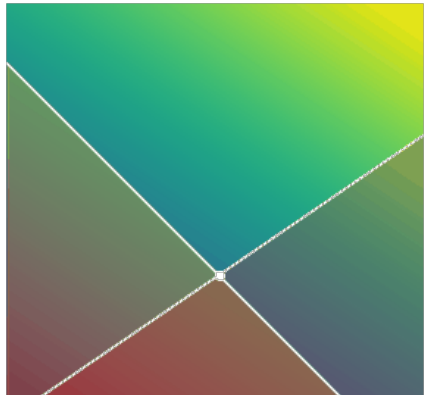
PyPSA



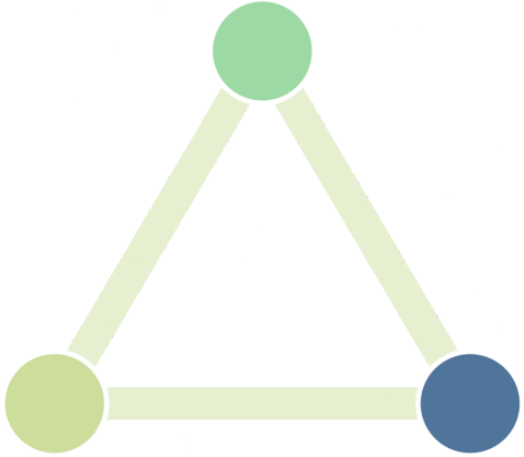
PyPSA-Eur



Linopy



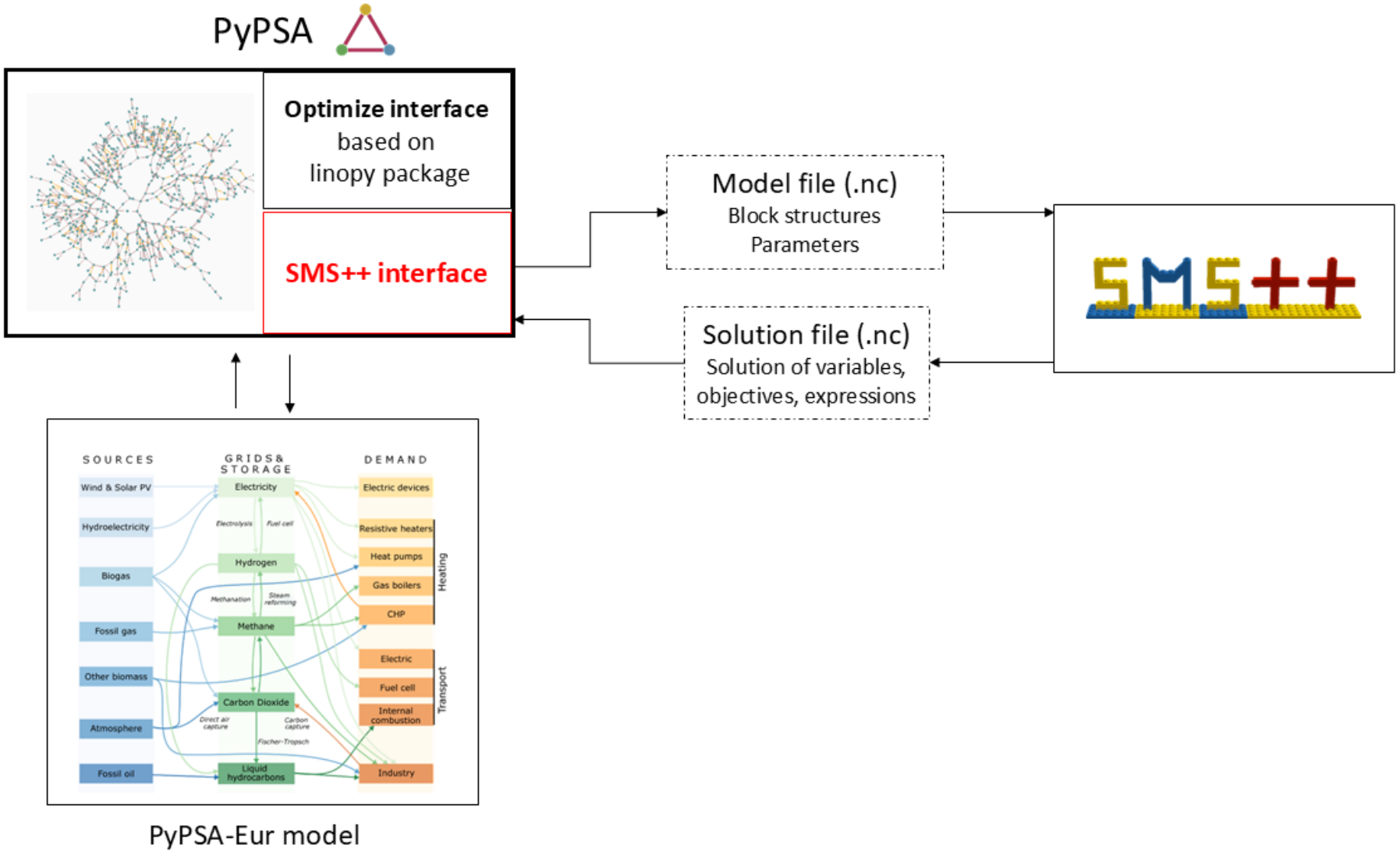
RESILIENT



[41] https://pypsa.org

[42] https://resilient-project.github.io

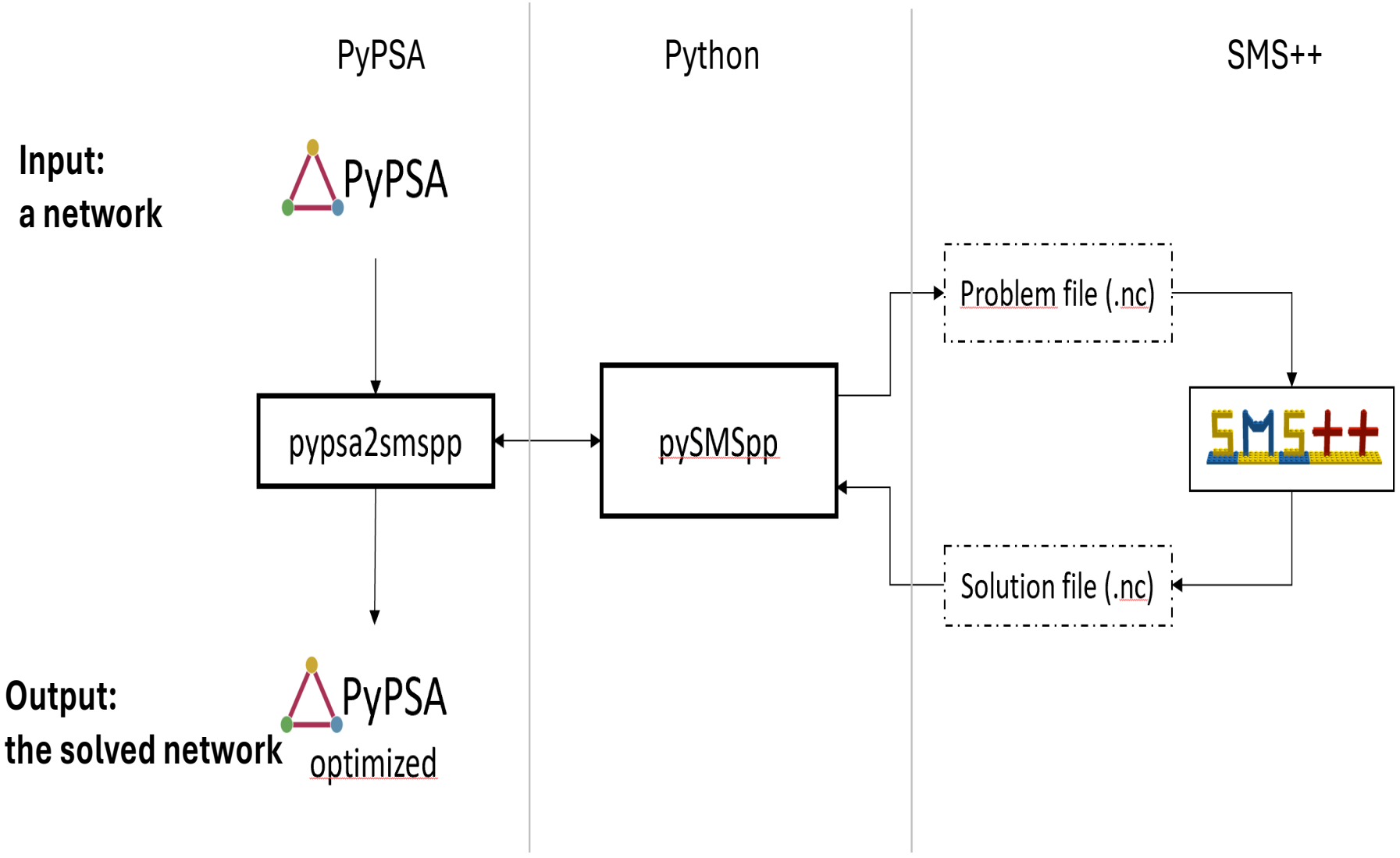
What the project wants to do



PyPSA-Eur model

- Modelling prowess of PyPSA + solution prowess of SMS++
- Basis for brand new PowerMax

A bit more in details

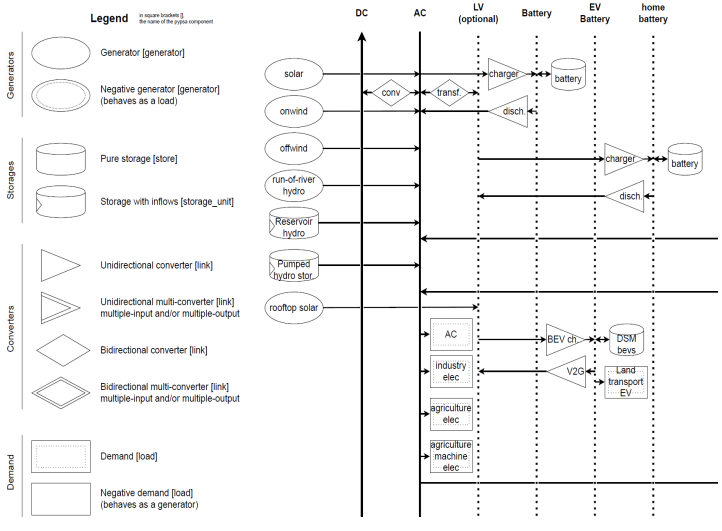


- Requires careful mapping of PyPSA network $\mapsto$ `SMS++ :Block`
- Especially important for freshly developed PyPSA stochastic extension

A glimpse of the mapping

Equation	Symbol	Generator	Link	Line	Storage unit	Store	Condition	Example
Size bound	SB	X	X	X	X	X		$\underline{G}_{i,r} \leq G_{i,r} \leq \overline{G}_{i,r}$
Modularity	MD	X	X	X	X	X		$G_{i,r} = \underline{G}^{mod}_{i,r} n_{i,r}$
Power bound	PB	X	X	X	X	X		$\underline{g}_{i,r,t} G_{i,r} \leq g_{i,r,t} \leq \overline{g}_{i,r,t} G_{i,r}$
Power unit commitment	PB_{UC}	X	X				committable	$\delta_{i,r,t} \underline{g}_{i,r,t} G_{i,r} \leq g_{i,r,t} \leq \delta_{i,r,t} \overline{g}_{i,r,t} G_{i,r}$
Minimum time	MT	X	X					$\sum_{t'=t}^{t+T^{min_{up}}} \delta_{k,t'} \geq T_{min_{up}} (\delta_{k,t} - \delta_{k,t-1})$
Total energy produced	PSUM	X	X					$E^{sum}_{min} \leq \sum_{t \in T} w^G_t g_{i,r,t} \leq E^{sum}_{max}$
Start up/shut down cost	SC	X	X					$suc_{k,t} \geq suc_k (\delta_{k,t} - \delta_{k,t-1})$
Rump up/down	RUD	X	X					$(g_{i,r,t} - g_{i,r,t-1}) \leq ru_{i,r} G_{i,r}$
Kirchhof's law	KL			X				$\sum_l C_{l,c} x_l p_{l,t} = 0$
Line losses	LL			X				$P^{loss}_l = \alpha_l + \beta_l p_{l,t}$
Energy storage level	ESL				X	X		$e_{i,s,t} = e_{i,s,t-1} + w^S_t h_{i,s,t}$
Energy storage bound	ESB				X	X		$0 \leq e_{i,s,t} \leq E_{i,s} e^{max}_{i,s}$
Initial energy level	IEL_S				X	X		$e_{i,s,0} = e_{i,s,init}$
Cyclic energy level	IEL_{CS}				X	X	cyclic_state_of_charge	$e_{i,s,0} = e_{i,s,\|T\|}$

	A	B	C	D	E
1	Technology name	Category	Physical compone	Option	Carrier
2	co2 atmosphere	co2	N		co2
3	Co2 storage	co2	Y		co2 stored
4	Sequestration link	co2	N		co2 sequestred
5	Sequestration store (e.g. underground)	co2	Y/N		co2 sequestred
6	CO2 vent co2 from storages	co2	?	co2_vent	co2 vent
7	CO2 pipelines	co2	Y	co2_network	CO2 pipeline
8	Allam (gas) cycle	electricity	Y	allam	allam
9	Direct Air Capture	co2	Y	dac	co2
10	Conventional generators	electricity	Y	conventional_generation	electricity
11	Haber-Bosch process	ammonia	Y	ammonia	Haber-Bosch
12	Ammonia cracker	ammonia	Y	ammonia	ammonia cracker
13	Ammonia storage	ammonia	Y	ammonia	ammonia store
14	Electricity distribution	electricity	Y	electricity_distribution_grid	low voltage
15	rooftop solar	electricity	Y	electricity_distribution_grid	solar rooftop



- . . . but no-one needs bother besides us (or new features developers)

What it means to you

- Let's see it working

- Let's see it working
- You can use it to be graded:
  - write your `netCDF` file from scratch with `pySMSpp`, solve it with `SMS++`
  - write a `PyPSA` network and have it solved by `SMS++` via `pypsa2smspp`

What it means to you

- Let's see it working
- You can use it to be graded:
  - write your `netCDF` file from scratch with `pySMSpp`, solve it with `SMS++`
  - write a `PyPSA` network and have it solved by `SMS++` via `pypsa2smspp`
- No need for anything realistic and / or large
- General-purpose solvers perfectly OK
- But if you want to scale higher, who am I to say no?
- If anyone interested, I'm here
- If you want more "theoretical" assignments, it can be done

Outline

Decomposition-aware modelling systems and solvers

SMS++ : design goals

SMS++ : basic ideas

Many other necessary pieces

SMS++ : existing Block and Solver

More advanced Block / Solver

SMS++ and beyond: (some of) the missing pieces

Conclusions (for good)

Conclusions (Part V)

- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- Could become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)

Conclusions (Part V)

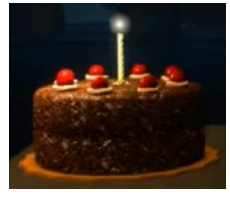
- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- Could become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)
- Hefty, very likely rather unrealistic, sough-after impacts:
  - improve collaboration and code reuse, reduce huge code waste (I ♡ coding, breaks my ♡)
  - significantly increase the addressable market of decomposition
  - a much-needed step towards higher uptake of parallel methods
  - the missing marketplace for specialised solution methods

Conclusions (Part V)

- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- Could become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)
- Hefty, very likely rather unrealistic, sough-after impacts:
  - improve collaboration and code reuse, reduce huge code waste (I ♡ coding, breaks my ♡)
  - significantly increase the addressable market of decomposition
  - a much-needed step towards higher uptake of parallel methods
  - the missing marketplace for specialised solution methods
- Not the only game in town, but aiming for a big slice of the cake

Conclusions (Part V)

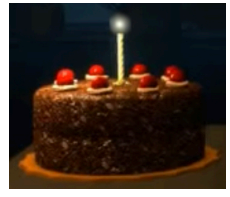
- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- Could become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)
- Hefty, very likely rather unrealistic, sough-after impacts:
  - improve collaboration and code reuse, reduce huge code waste (I ♡ coding, breaks my ♡)
  - significantly increase the addressable market of decomposition
  - a much-needed step towards higher uptake of parallel methods
  - the missing marketplace for specialised solution methods
- Not the only game in town, but aiming for a big slice of the cake which (hopefully) is not a lie 😉 😉 😉



- Trying hard to increase the outreach via the PyPSA interface

Conclusions (Part V)

- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- Could become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)
- Hefty, very likely rather unrealistic, sough-after impacts:
  - improve collaboration and code reuse, reduce huge code waste (I ♡ coding, breaks my ♡)
  - significantly increase the addressable market of decomposition
  - a much-needed step towards higher uptake of parallel methods
  - the missing marketplace for specialised solution methods
- Not the only game in town, but aiming for a big slice of the cake which (hopefully) is not a lie 😉 😉 😉



- Trying hard to increase the outreach via the PyPSA interface
- A community-building effort as much as a software product:

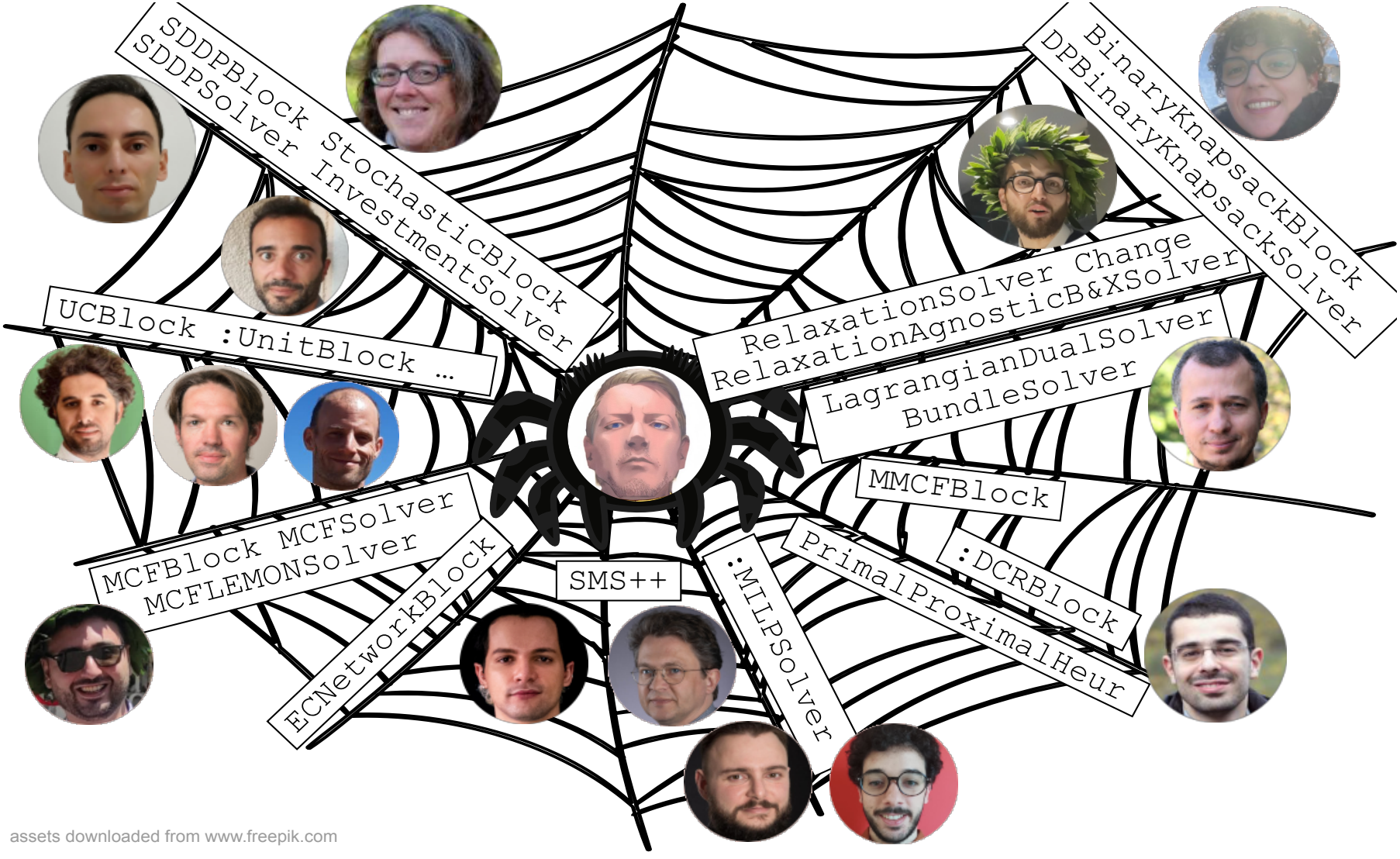
Conclusions (Part V)

- SMS++ is there, actively developed
- Perhaps already useful for some fringe use cases
- Could become really useful after having attracted mindshare, self-reinforcing loop (very hard to start)
- Hefty, very likely rather unrealistic, sough-after impacts:
  - improve collaboration and code reuse, reduce huge code waste (I ♡ coding, breaks my ♡)
  - significantly increase the addressable market of decomposition
  - a much-needed step towards higher uptake of parallel methods
  - the missing marketplace for specialised solution methods
- Not the only game in town, but aiming for a big slice of the cake which (hopefully) is not a lie 😉 😉 😉



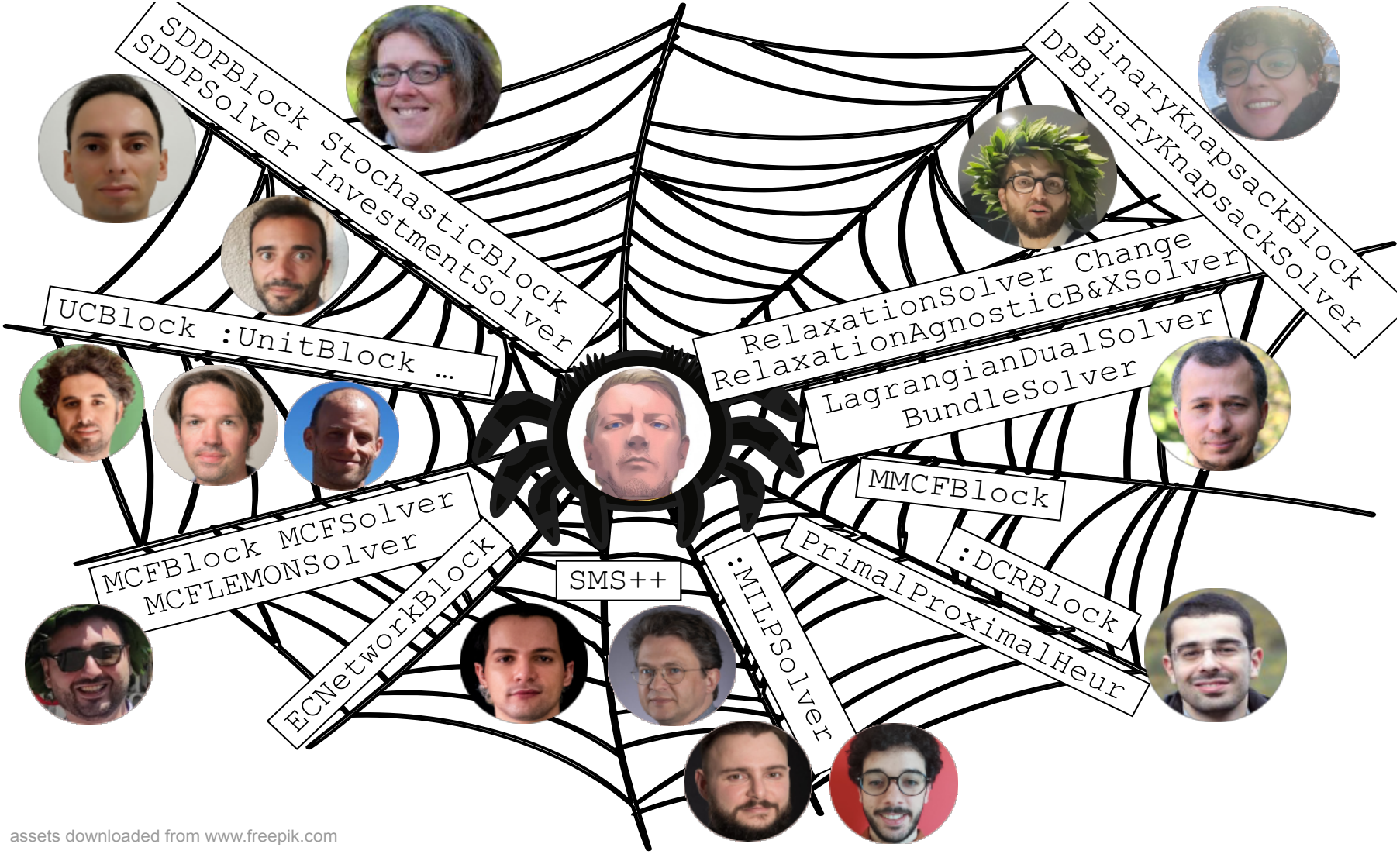
- Trying hard to increase the outreach via the PyPSA interface
- A community-building effort as much as a software product: is it working?

. . . see it for yourself



- Not (yet) a thriving community, but labouring to reach critical mass

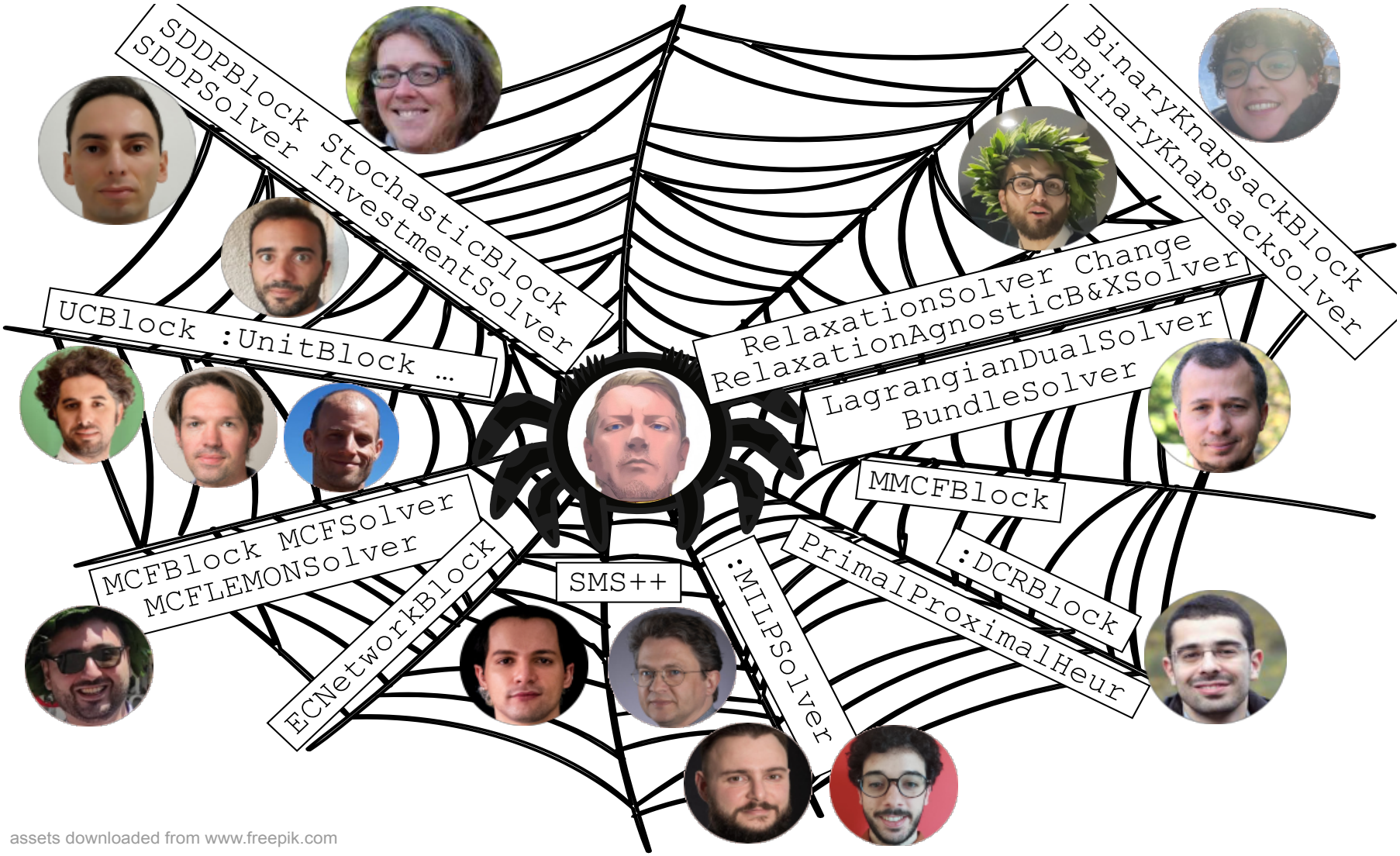
. . . see it for yourself



- Not (yet) a thriving community, but labouring to reach critical mass
- Plenty of room in the Hotel California



. . . see it for yourself



- Not (yet) a thriving community, but labouring to reach critical mass
- Plenty of room in the spiderweb, be my guest



Conclusions overall

- Energy optimization is difficult

Conclusions overall

- Energy optimization is difficult . . . but there are $\neq$ kinds of "difficult"
- Difficult because hard, difficult because large, or most likely both

Conclusions overall

- Energy optimization is difficult . . . but there are $\neq$ kinds of "difficult"
- Difficult because hard, difficult because large, or most likely both
- Chasing a nontrivial honourable compromise between:
  - what you would really like to solve
  - what it is reasonably possible to solve

Conclusions overall

- Energy optimization is difficult . . . but there are $\neq$ kinds of "difficult"
- Difficult because hard, difficult because large, or most likely both
- Chasing a nontrivial honourable compromise between:
  - what you would really like to solve
  - what it is reasonably possible to solve
- It is easy to err by excess of optimism, but it is also quite possible to err by excess of pessimism
- Some crazy nuts relish with being given to solve very hard problems 
- This may require hellishly complex software, but this is little of your concern
- Meanwhile, general-purpose tools keep improving and can solve though stuff

Conclusions overall

- Energy optimization is difficult . . . but there are $\neq$ kinds of "difficult"
- Difficult because hard, difficult because large, or most likely both
- Chasing a nontrivial honourable compromise between:
  - what you would really like to solve
  - what it is reasonably possible to solve
- It is easy to err by excess of optimism, but it is also quite possible to err by excess of pessimism
- Some crazy nuts relish with being given to solve very hard problems 
- This may require hellishly complex software, but this is little of your concern
- Meanwhile, general-purpose tools keep improving and can solve though stuff
- Overall: get a sense of what can be done easily when the sun shines

Conclusions overall

- Energy optimization is difficult . . . but there are $\neq$ kinds of "difficult"
- Difficult because hard, difficult because large, or most likely both
- Chasing a nontrivial honourable compromise between:
  - what you would really like to solve
  - what it is reasonably possible to solve
- It is easy to err by excess of optimism, but it is also quite possible to err by excess of pessimism
- Some crazy nuts relish with being given to solve very hard problems 
- This may require hellishly complex software, but this is little of your concern
- Meanwhile, general-purpose tools keep improving and can solve though stuff
- Overall: get a sense of what can be done easily when the sun shines + make an optimization friend for when the rain starts (since then it pours)

Conclusions overall

- Energy optimization is difficult . . . but there are $\neq$ kinds of "difficult"
- Difficult because hard, difficult because large, or most likely both
- Chasing a nontrivial honourable compromise between:
  - what you would really like to solve
  - what it is reasonably possible to solve
- It is easy to err by excess of optimism, but it is also quite possible to err by excess of pessimism
- Some crazy nuts relish with being given to solve very hard problems 
- This may require hellishly complex software, but this is little of your concern
- Meanwhile, general-purpose tools keep improving and can solve though stuff
- Overall: get a sense of what can be done easily when the sun shines
  + make an optimization friend for when the rain starts (since then it pours)
  = you can have and let have lots of fun (ask Davide, Giovanni, Fabrizio . . .)