It's a Bird... It's a Plane... It's SMS++

Antonio Frangioni

Dipartimento di Informatica, Università di Pisa

MINOA ESR Days 2021
March 4, 2021

Outline

Why bother?

- Mainly a selfish reason and an altruistic one

Why bother?

- Mainly a selfish reason and an altruistic one
- Selfish reason: I've made decomposition stuff for many years, but (almost) nobody cares

Why bother?

- Mainly a selfish reason and an altruistic one
- Selfish reason: I've made decomposition stuff for many years, but (almost) nobody cares
- Altruistic version of the selfish reason: decomposition may be useful for many, but it's too difficult to do in practice

Why bother?

- Mainly a selfish reason and an altruistic one
- Selfish reason: I've made decomposition stuff for many years, but (almost) nobody cares
- Altruistic version of the selfish reason: decomposition may be useful for many, but it's too difficult to do in practice
- Altruistic reason: stop a lot of good programming going to waste

Why bother?

- Mainly a selfish reason and an altruistic one
- Selfish reason: I've made decomposition stuff for many years, but (almost) nobody cares
- Altruistic version of the selfish reason: decomposition may be useful for many, but it's too difficult to do in practice
- Altruistic reason: stop a lot of good programming going to waste
- Programming is hard, good programming is harder, optimization is hard, good programming for optimization is extremely hard
- Typically done early on in the career (you)
- Does not "pay" much career-wise (wrong)
- Many months/years of your life go down the drain because developing, maintaining and supporting a good package is not "cost-effective"

Why bother?

- Mainly a selfish reason and an altruistic one
- Selfish reason: I've made decomposition stuff for many years, but (almost) nobody cares
- Altruistic version of the selfish reason: decomposition may be useful for many, but it's too difficult to do in practice
- Altruistic reason: stop a lot of good programming going to waste
- Programming is hard, good programming is harder, optimization is hard, good programming for optimization is extremely hard
- Typically done early on in the career (you)
- Does not "pay" much career-wise (wrong)
- Many months/years of your life go down the drain because developing, maintaining and supporting a good package is not "cost-effective"
- Selfish version of the altruistic reason: if all this work was readily available decomposition would make a lot more sense, and my own work would be more useful

Outline

Decomposition-aware modelling systems

- Decomposition is complex, but so is any Branch-and-X
- Need general-purpose efficient decomposition software:
  - Cplex does Benders', structure automatic or user hints
  - SCIP[1] does B&C&P (one-level D-W), pricing & reformulation up to the user (plugins)
  - GCG[1] extends SCIP with automatic and user-defined (one-level) D-W and recently also a generic (one-level) Benders' approach[2]
  - D-W approaches for two-stage stochastic programs are implemented in DDSIP[3] and PIPS[4], the latter interfaced with StructJuMP[5]
  - The BaPCoD B&C&P code has been used to develop Coluna.jl[6], doing one-level D-W and (alpha) Benders', multi-level planned
- No multi-level C++, so we started one

---

[1] https://scipopt.org, https://gcg.or.rwth-aachen.de

[2] Maher "Implementing the Branch-and-Cut approach for a general purpose Benders' decomposition framework" *EJOR*, 2021

[3] https://github.com/RalfGollmer/ddsip

[4] https://github.com/Argonne-National-Laboratory/PIPS

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

[6] https://github.com/atoptima/Coluna.jl

Outline

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

Open source (LGPL3)

Public as of February 8, but some 8+ years in the making

What SMS++ is

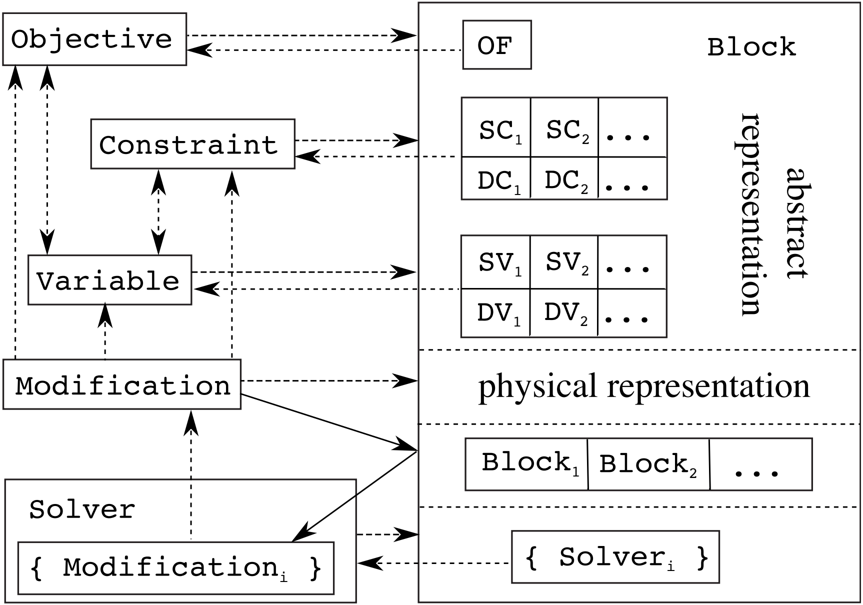
- A core set of C++-17 classes implementing a modelling system that:
  - explicitly supports the notion of Block ≡ nested structure
  - separately provides "semantic" information from "syntactic" details (list of constraints/variables ≡ one specific formulation among many)
  - allows exploiting specialised 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

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 0.4, lots of further development ahead, significant changes in interfaces not ruled out, actually expected (although current `Block` / `Solver` very thoroughly tested)
- Interfaced with many solvers: only `Cplex`, `SCIP`, `MCFClass`, `StOpt` (although the list should hopefully grow)

Outline

A Crude Schematic



Block

- Block = abstract class representing the general concept of "a (part of a) mathematical model with a well-understood identity"
- Each :Block a model with specific structure (e.g., MCFBlock:Block = a Min-Cost Flow problem)
- Physical representation of a Block: whatever data structure is required to describe the instance (e.g., G, b, c, u)
- Possibly alternative abstract representation(s) of a Block:
  - one Objective (but possibly vector-valued)
  - any # of groups of (static) Variable
  - any # of groups of std::list of (dynamic) Variable
  - any # of groups of (static) Constraint
  - any # of groups of std::list of (dynamic) Constraint

  groups of Variable/Constraint can be single (std::list) or std::vector (...) or boost::multi_array
- Any # of sub-Blocks (recursively), possibly of specific type (e.g., Block::MMCFBlock has k Block::MCFBlock inside)

Variable

- Abstract concept, thought to be extended (a matrix, a function, . . .)
- Does not even have a value
- Knows which `Block` it belongs to
- Can be fixed and unfixed to/from its current value (whatever that is)
- Influences a set of `Constraint`/`Objective`/`Function` (actually, a set of `ThinVarDepInterface`)
- Fundamental design decision: “name” of a `Variable` = its memory address $\Longrightarrow$ copying a `Variable` makes a different `Variable` $\Longrightarrow$ dynamic `Variables` always live in `std::lists`
- `VariableModification:Modification` (fix/unfix)

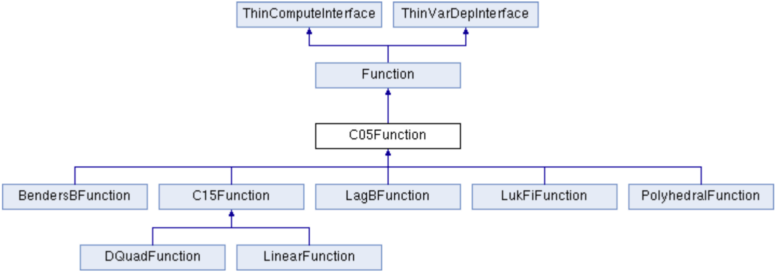
Constraint

- Abstract concept, thought to be extended (any algebraic constraint, a matrix constraint, a PDE constraint, bilevel program, ...)
- Depends from a set of `Variable` (`:ThinVarDepInterface`)
- Either satisfied or not by the current value of the `Variable`, checking it possibly costly (`:ThinComputeInterface`)
- Knows which `Block` it belongs to
- Can be relaxed and enforced
- Fundamental design decision: "name" of a `Constraint` = its memory address $\Longrightarrow$ copying a `Constraint` makes a different `Constraint` $\Longrightarrow$ dynamic `Constraint`s always live in `std::list`s
- `ConstraintModification:Modification` (relax/enforce)

Objective

- Abstract concept, does not specify its return value (vector, set, ...)
- Either minimized or maximized
- Depends from a set of `Variable` (`:ThinVarDepInterface`)
- Must be evaluated w.r.t. the current value of the `Variable`, possibly a costly operation (`:ThinComputeInterface`)
- `RealObjective:Objective` implements "value is an extended real"
- Knows which `Block` it belongs to
- Same fundamental design decision ...
  (but there is no such thing as a dynamic `Objective`)
- `ObjectiveModification:Modification` (change verse)

Function



- Real-valued `Function`
- Depends from a set of `Variable` (`:ThinVarDepInterface`)
- Must be evaluated w.r.t. the current value of the `Variable`, possibly a costly operation (`:ThinComputeInterface`)
- Approximate computation supported in a quite general way[56] (since `:ThinComputeInterface`, and that does)
- `FunctionModification[Variables]` for "easy" changes $\Longrightarrow$ reoptimization (shift, adding/removing "quasi separable" `Variable`)

C05Function and C15Function

- C05Function/C15Function deal with $1^{st}/2^{nd}$ order information (not necessarily continuous)
- General concept of "linearization" (gradient, convex/concave subgradient, Clarke subgradient, ...)
- Multiple linearizations produced at each evaluation (local pool)
- Global pool of linearizations for reoptimization:
  - convex combination of linearizations
  - "important linearization" (at optimality)
- C05FunctionModification[Variables/LinearizationShift] for "easy" changes $\Longrightarrow$ reoptimization (linearizations shift, some linearizations entries changing in simple ways)
- C15Function supports (partial) Hessians
- Arbitrary hierarchy of :Function possible/envisioned, any one that makes sense for application and/or solution method

Closer to the ground

- `ColVariable:Variable`: "value = one single real" (possibly $\in \mathbb{Z}$)
- `RowConstraint:Constraint`: "$l \leq$ a real $\leq u$" $\Longrightarrow$ has dual variable (single real) attached to it
- `OneVarConstraint:RowConstraint`: "a real" = a single `ColVariable` $\equiv$ bound constraints
- `FRowConstraint:RowConstraint`: "a real" given by a `Function`
- `FRealObjective:RealObjective`: "value" given by a `Function`
- `LinearFunction:Function`: a linear form in `ColVariable`
- `DQuadFunction:Function`: a separable quadratic form
- Many things missing (`AlgebraicFunction`, `DenseLinearFunction`, `Matrix/VectorVariable`, ...)

Block and Solver

- Any # of Solver attached to a Block to solve it
- :Solver for a specific :Block can use the physical representation
  $\Longrightarrow$ no need for explicit Constraint
  $\Longrightarrow$ abstract representation of Block only constructed on demand
- However, Variable are always present to interface with Solver (this may change thanks to methods factory)
- A general-purpose Solver uses the abstract representation
- Dynamic Variable/Constraint can be generated on demand (user cuts/lazy constraints/column generation)
- For a Solver attached to a Block:
  - Variable not belonging to the Block are constants
  - Constraint not belonging to the Block are ignored

  (belonging = declared there or in any sub-Block recursively)
- Objective of sub-Blocks summed to that of father Block if has same verse, otherwise min/max

Solver

- Solver = interface between a Block and algorithms solving it
- Each Solver attached to a single Block, from which it picks all the data, but any # of Solver can be attached to the same Block
- Solutions are written directly into the Variable of the Block
- Individual Solver can be attached to sub-Block of a Block
- Tries to cater for all the important needs:
  - optimal and sub-optimal solutions, provably unbounded/unfeasible
  - time/resource limits for solutions, but restarts (reoptimization)
  - any # of multiple solutions produced on demand
  - lazily reacts to changes in the data of the Block via Modification
- Slanted towards RealObjective (≈optimality = up/low bounds)
- CDASolver:Solver is "Convex Duality Aware": bounds are associated to dual solutions (possibly, multiple)
- Provides general events mechanism (ThinComputeInterface does)

Block and Modification

- Most Block components can change, but not all:
  - set of sub-Block
  - # and shape of groups of Variable/Constraint
- Any change is communicated to each interested Solver (attached to the Block or any of its ancestor) via a Modification object
- anyone_there() $\equiv \exists$ interested Solver (Modification needed)
- However, two different kinds of Modification (what changes):
  - physical Modification, only specialized Solver concerned
  - abstract Modification, only Solver using it concerned
- Abstract Modification used to keep both representations in sync
  $\Longrightarrow$ a single change may trigger more than one Modification
  $\Longrightarrow$ concerns_Block() mechanism to avoid this to repeat
  $\Longrightarrow$ parameter in changing methods to avoid useless Modification
- Specialized Solver disregard abstract Modification and vice-versa
- A Block may refuse to support some changes (explicitly declaring it)

Modification

- Almost empty base class, then everything has its own derived ones
- Heavy stuff can be attached to a `Modification` (e.g., added/deleted dynamic `Variable`/`Constraint`)
- Each `Solver` has the responsibility of cleaning up its list of `Modification` (smart pointers → memory eventually released)
- `Solver` supposedly reoptimize to improve efficiency, which is easier if you can see all list of changes at once (lazy update)
- `GroupModification` to (recursively) pack many `Modification` together ⟹ different "channels" in `Block`
- `Modification` processed in the arrival order to ensure consistency
- A `Solver` may optimize the changes (`Modifications` may cancel each outer out . . .), but its responsibility

Support to (coarse-grained) Parallel Computation

- `Block` can be (r/w) `lock()`-ed and `read_lock()`-ed
- `lock()`-ing a `Block` automatically `lock()`s all inner `Block`
- `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)
- Similar mechanism for `read_lock()`, any # of concurrent reads
- Write starvation not handled yet
- A `Solver` can be "lent an ID" (solving an inner `Block`)
- The list of `Modification` of `Solver` is under an "active guard" (`std::atomic`)
- Distributed computation under development, can exploit general serialize/deserialize `Block` capabilities, Cray/HPE "Fugu" framework

Solution

- Block produces Solution object, possibly using its sub-Blocks'
- Solution can read() its own Block and write() itself back
- Solution is Block-specific rather than Solver-specific
- Solution may save dual information
- Solution may save only a specific subset of primal/dual information
- Linear combination of Solution supported $\Longrightarrow$ "less general" Solution may (automatically) convert in "more general" ones
- Like Block, Solution are tree-structured complex objects
- ColVariableSolution:Solution uses the abstract representation of any Block that only have (std::vector or boost::multi_array of) (std::list of) ColVariables to read/write the solution
- RowConstraintSolution:Solution same for dual information (RowConstraint), ColRowSolution for both

Configuration

- Block a tree-structured complex object $\Longrightarrow$ Configuration for them a (possibly) tree-structured complex object
- But also SimpleConfiguration<T>:Configuration (T an int, a double, a std::pair<>, ...)
- [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

R^3Block

- 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 R^3Blocks :Block-specific, a :Configuration needed
- R^3Block completely independent (new Variable/Constraint), useful for algorithmic purposes (branch, fix, solve, ...)
- Solution of R^3Block useful to Solver for original Block: map_back_solution() (best effort in case of dynamic Variable)
- Sometimes keeping R^3Block 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 R^3Block
- :Block is in charge of all this, thus decides what it supports

A lot of other support stuff

- All tree-structured complex objects (`Block`, `Configuration`, . . .) and `Solver` have an (almost) automatic factory
- All tree-structured complex objects (. . .) have methods to serialize/deserialize themselves to `netCDF` files
- A methods factory for changing the physical representation without knowing of which `:Block` it exactly is (standardised interface)
- `AbstractBlock` for constructing a model a-la algebraic language, can be derived for "general `Block` + specific part"
- `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

Main Existing :Block

- MCFBlock/MMCFBlock: single/multicommodity flow (p.o.c.)
- UCBlock for UC, abstract UnitBlock with several concrete (ThermalUnitBlock, HydroUnitBlock, ...), abstract NetworkBlock with a few concrete (DCNetworkBlock)
- LagBFunction:{C05Function,Block} transforms any Block (with appropriate Objective) into its dual function
- BendersBFunction:{C05Function,Block} transforms any Block (with appropriate Constraint) into its value function
- StochasticBlock implements realizations of scenarios into any Block (using methods factory)
- SDDPBlock represents multi-stage stochastic programs suitable for Stochastic Dual Dynamic Programming

Main "Basic" :Solver

- MCFSolver: templated p.o.c. wrapper to MCFClass[7] for MCFBlock
- DPSolver for ThermalUnitBlock (still needs serious work)
- MILPSolver: constructs matrix-based representation of any "LP" Block: ColVariable, FRowConstraint, FRealObjective with LinearFunction or DQuadFunction
- CPXMILPSolver:MILPSolver and SCIPMILPSolver:MILPSolver wrappers for Cplex and SCIP (to be improved)
- BundleSolver:CDASolver: SMS++-native version of[8] (still shares some code, dependency to be removed), optimizes any (sum of) C05Function, several (but not all) state-of-the-art tricks
- SDDPSolver: wrapper for SDDP solver StOpt[9] using StochasticBlock, BendersBFunction and PolyhedralFunction
- SDDPGreedySolver: greedy forward simulator for SDDPBlock

[7] https://github.com/frangio68/Min-Cost-Flow-Class

[8] https://gitlab.com/frangio68/ndosolver_fioracle_project

[9] https://gitlab.com/stochastic-control/StOpt

Our Masterpiece: LagrangianDualSolver

- Works for any Block with natural block-diagonal structure: no Objective or Variable, all Constraint linking the inner Block
- Using LagBFunction stealthily constructs the Lagrangian Dual w.r.t. linking Constraint, R^3B-ing or "stealing" the inner Block
- Solves the Lagrangian Dual with appropriate CDASolver (e.g., but not necessarily, BundleSolver), provides dual and "convexified" solution in original Block
- Can attach LagrangianDualSolver and (say) :MILPSolver to same Block, solve in parallel!
- Weeks of work in days/hours (if Block of the right form already)
- Hopefully soon BendersDecompositionSolver (crucial component BendersBFunction existing and tested)
- Multilevel nested parallel heterogeneous decomposition by design (but I'll believe it when I'll see it running)

Outline

The many things that we do not have (yet)

- A relaxation-agnostic Branch-and-X `Solver` (could recycle `OOBB`?)
- 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 (`GuRoBi`, `OSISolverInterface`, `Couenne`, 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

Outline

Conclusions and (a lot of) future work

- 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 and (a lot of) future work

- 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
  - a step towards a reformulation-aware modelling system[10]

Conclusions and (a lot of) future work

- 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
  - a step towards a reformulation-aware modelling system[10]
- Luckily not the only game in town, but aiming for a slice of the cake

Conclusions and (a lot of) future work

- 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
  - a step towards a reformulation-aware modelling system[10]
- Luckily not the only game in town, but aiming for a slice of  which is not a lie ☺ ☺ ☺

Conclusions and (a lot of) future work

- 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
  - a step towards a reformulation-aware modelling system[10]
- Luckily not the only game in town, but aiming for a slice of  which is not a lie 😉 😉 😉
- Lots of fun to be had, all contributions welcome

[10] F., Perez Sanchez "Transforming Mathematical Models Using Declarative Reformulation Rules" *LNCS*, 2011

Acknowledgements

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 773897

Copyright © Università di Pisa 2021, all rights reserved.

This document may not be copied, reproduced, or modified in whole or in part for any purpose without written permission from the PLAN4RES Consortium. In addition, an acknowledgement of the authors of the document and all applicable portions of the copyright notice must be clearly referenced.

This document may change without notice.

The content of this document only reflects the author's views. The European Commission / Innovation and Networks Executive Agency is not responsible for any use that may be made of the information it contains.