# Artificial Intelligence Techniques for Automatic Reformulation and Solution of Structured Mathematical Models 

Luis Perez Sanchez<br>SUPERVISOR<br>Antonio Frangioni<br>Referee<br>Andreas Grothey

October 15, 2010


Luis Perez Sanchez


Antonio Frangioni


#### Abstract

Complex, hierarchical, multi-scale industrial and natural systems generate increasingly large mathematical models. Practitioners are usually able to formulate such models in their "natural" form; however, solving them often requires finding an appropriate reformulation to reveal structures in the model which make it possible to apply efficient, specialized approaches. The search for the "best" formulation of a given problem, the one which allows the application of the solution algorithm that best exploits the available computational resources, is currently a painstaking process which requires considerable work by highly skilled personnel. Experts in solution algorithms are required for figuring out which (formulation, algorithm) pair is better used, considering issues like the appropriate selection of the several obscure algorithmic parameters that each solution methods has. This process is only going to get more complex, as current trends in computer technology dictate the necessity to develop complex parallel approaches capable of harnessing the power of thousands of processing units, thereby adding another layer of complexity in the form of the choice of the appropriate (parallel) architecture. All this renders the use of mathematical models exceedingly costly and difficult for many potentially fruitful applications. The I-DARE environment, proposed in this Thesis, aims at devising a software system for automatizing the search for the best combination of (re)formulation, solution algorithm and its parameters (comprised the computational architecture), until now a firm domain of human intervention, to help practitioners bridging the gap between mathematical models cast in their natural form and existing solver systems. I-DARE deals with deep and challenging issues, both from the theoretical and from an implementative viewpoint: 1) the development of a language that can be effectively used to formulate large-scale structured mathematical models and the reformulation rules that allow to transform a formulation into a different one; 2) a core subsystem capable of automatically reformulating the models and searching in the space of (formulations, algorithms, configurations) able to "the best" formulation of a given problem; 3) the design of a general interface for numerical solvers that is capable of accommodate and exploit structure information. To achieve these goals I-DARE will propose a sound and articulated integration of different programming paradigms and techniques like, classic Object-Oriented programing and Artificial Intelligence (Declarative Programming, Frame-Logic, Higher-Order Logic, Machine Learning). By tackling these challenges, I-dare may have profound, lasting and disruptive effects on many facets of the development and deployment of mathematical models and the corresponding solution algorithms.


## Acknowledgments

I would like to thank all the people that, in one way or another, helped me to walk across this beautiful and sometimes steep road. In particular, infinite thanks to my supervisor Antonio Frangioni. Without his patience, understanding and clarifying ideas, the realization of this work would have been impossible.

Thanks to Giorgio Levi for suggesting me to apply for the PhD in the first place and encouraging and supporting me throughout all the process. Thanks to Pierpaolo Degano and Ilaria Fierro, for an excellent PhD school organization and for giving us (foreign PhD students) full support during all these three years. Thanks to Giorgio Gallo for the crucial advices at each stage of the Thesis. Thanks to Andrea Maggiolo, Fabrizio Luccio and Linda Pagli for all the help and support since the very beginning.

Thanks to Luciano Garcia Garrido for introducing me to the Logic and Artificial Intelligence and for always motivating me to explore new and interdisciplinary points of view.

Thanks to Leo Liberti for the discussions regarding the experiments that helped me to build the methodology applied in this Thesis.

Thanks to Basuki, Flavio and Jorge for being three excellent PhD colleagues and friends and for all the moments together that made this experience a really nice one. Thanks to Sara, Fabrizio, and Sergio for the "intense" Bowling competitions. Thanks to Cristina and Daniela for always being caring friends even when we barely knew each other.

Vorrei inoltre ringraziare il Furnari, la Roglia, il Cataldi e il resto del team Torino, per uno spettacolare Bertinoro e per tutti bei momenti che abbiamo passato insieme. Grazie a Sara Capecchi per essere stata sempre dalla nostra parte e averci sempre appoggiato.

Grazie al team Cuneo: Chiara, Damiano, Elisa, Seutta, Valeria, Luca, Anna e Diego, per avermi accettato sempre come parte del gruppo, per le tante chiacchierate (anche quando io parlavo sempre in inglese). Grazie a mia "cognata" Mara, per la compagnia e l'amicizia, ma soprattutto per le grasse risate. Grazie a Danila e Diego per essere diventati dei fratelli per me, per le tante serate a giocare a domino mentre si ascoltavano le selezioni musicali da YouTube, e per tutti i preziosi momenti insieme.

Grazie a Carla e Gra per essere dei suoceri meravigliosi, per avermi accolto nella loro famiglia come un figlio, per tutto il supporto e la comprensione, e per i meravigliosi piatti che mi avete fatto assaggiare.

Gracias al Luisda, Pabli, Kmilo, Batista, Jose Carlos, Tomy, Mija, por todos los valiosos momentos, desde las noches en el cubículo 3 del C5, hasta las clases en la UH, las inolvidables sesiones en UH-SIS y podría seguir adelante.

Gracias a mi hermanita Thaizel por la importantísima compañía durante todo este tiempo en Pisa, fundamentalmente durante el difícil inicio.

A los mejores padres que un hijo pueda desear, gracias a los dos por haberme apoyado siempre en mis decisiones, aún cuando eran decisiones difíciles. Gracias por todos los utiles consejos y por todo el amor que me han dado siempre. Por supuesto gracias tambien a la super abuela Zenaida por todo su cariño (y espectaculares garbanzos).

Elena, mi è molto difficile fare soltanto una frase, GRAZIE DI TUTTO!

## Contents

Introduction ..... i
I. 1 Modeling and solving with structures - Motivations ..... ii
I. 2 The proposal and further motivations ..... iii
I. 3 I-DARE - Overview ..... vii
I. 4 Thesis structure ..... ix
1 I-DARE(lib) - the structure library ..... 1
1.1 Basic components ..... 1
1.1.1 Parameter Types ..... 2
1.2 Structure Classes ..... 2
1.2.1 Leaf Problem Class ..... 3
1.2.2 Block Class ..... 6
1.2.3 Block - Formal definitions ..... 7
1.3 Some Structure Class examples ..... 10
1.4 Discussion ..... 12
2 I-DARE(im) - base modeling environment ..... 13
2.1 Dimensions ..... 13
2.1.1 Local dimensions ..... 13
2.2 Variables and Constants ..... 14
2.2.1 Bounding ..... 15
2.2.2 Property well-formedness ..... 15
2.2.3 Indexing ..... 16
2.2.4 Expressions ..... 17
2.3 Scalars and Vectors ..... 20
2.4 Leaf Problem ..... 23
2.4.1 Local Leaf Problems ..... 24
2.5 Blocks ..... 26
2.6 Formulation ..... 29
2.7 Discussion ..... 31
3 I-DARE(ei) - the enhanced instance ..... 33
3.1 Instance Wrapper (IW) ..... 33
3.1.1 Global Data ..... 34
3.1.2 Data handler (DH) ..... 36
3.1.3 IW - Data Validation ..... 40
3.1.4 IW - Access methods ..... 41
3.2 Structured Instance Generation ..... 42
3.2.1 Meta Data File ..... 42
3.3 Discussion ..... 47
4 I-DARE(solve) - solving the model ..... 49
4.1 Structured Modeling and Solving Methods ..... 49
4.1.1 Some initial formulation ..... 49
4.1.2 Knapsack constraints ..... 50
4.1.3 Logical Formula ..... 54
4.1.4 Disjunctive Scheduling ..... 54
4.1.5 Product Loading ..... 55
4.2 Shared Variables - Blocks - Relaxation ..... 58
4.2.1 Returning to the Product Loading problem ..... 59
4.2.2 Tackling the other formulations ..... 60
4.3 Solvers' Tree ..... 61
4.4 Solvers and Structures ..... 63
4.4.1 Solvers exported and new Structure relations ..... 65
4.4.2 Configuration Templates ..... 67
4.4.3 Solution Generation ..... 69
4.4.4 The overall solution process ..... 69
5 I-DARE( t ) - the reformulation system ..... 71
5.1 Atomic Reformulation Rules ..... 71
5.1.1 Abstract structure classes ..... 72
5.1.2 Track Structures ..... 73
5.2 Algebraic ARR ..... 74
5.2.1 ARR $^{\Sigma}$ - Index declarations ..... 75
5.2.2 $\mathrm{ARR}^{\Sigma}$ - Dimension relations ..... 76
5.2.3 $\mathrm{ARR}^{\Sigma}-$ Mappings ..... 76
5.2.4 $\mathrm{ARR}^{\Sigma}-$ Fixed template items ..... 80
5.2.5 ARR $^{\Sigma}$ - Conditional expression ..... 80
5.2.6 $\mathrm{ARR}^{\sum}$ - examples ..... 81
5.2.7 Semantics ..... 83
5.3 Algorithmic ARR ..... 87
5.3.1 Semantics ..... 88
5.4 Selection domain and Reformulation Domain ..... 90
5.5 Discussion ..... 91
6 I-DARE(control) - best (Formulation, Solver, Configuration) ..... 93
6.1 Search Spaces ..... 93
6.1.1 Extended Model ..... 94
6.1.2 Solvers and Configurations ..... 94
6.2 Controlling the Search in the (Formulation, Solver, Configuration) Space ..... 94
6.2.1 Objective function computation ..... 95
6.2.2 Training and Meta-Learning ..... 99
6.2.3 The overall search process ..... 99
6.3 Experiments ..... 100
6.3.1 Further experimentation ..... 102
6.4 Discussion ..... 103
7 Combining structures and reformulations ..... 105
7.1 Structures ..... 105
7.1.1 Compositions ..... 107
7.2 Creating a model ..... 108
7.3 Reformulations ..... 109
7.3.1 ProdBC to MILP ..... 110
7.3.2 SAbs to Composition ..... 111
7.3.3 VAbs to LP ..... 111
7.3.4 SemiContinuous to MILP ..... 112
7.3.5 ProdCC to MILP ..... 114
7.3.6 SemiAssign to MILP ..... 114
7.3.7 Constraint to MILP ..... 115
7.3.8 OFMin to MILP ..... 115
7.3.9 IndComposition to MILP ..... 116
7.3.10 Composition to MILP ..... 117
7.4 Applying the $\mathrm{ARR}^{\Sigma_{\mathrm{s}}}$ to HCP ..... 119
8 More focused structures and reformulations ..... 123
8.1 Rounding Up previously defined structures ..... 123
8.1.1 Reformulations ..... 124
8.1.2 Reformulation Diagram ..... 129
8.2 Some Convex Structures ..... 130
8.2.1 Reformulations ..... 132
8.3 Quadratic Variants ..... 136
8.3.1 Reformulations ..... 137
8.3.2 Final diagram ..... 140
Conclusions ..... 143
C. 1 Perspectives on Deployment ..... 144
C. 2 I-DARE challenges ..... 146
Bibliography ..... 149
A Frame Logic - FLORA-2 ..... 159
A. 1 FL Syntax ..... 159
A.1.1 Alphabet ..... 159
A.1.2 Terms ..... 160
A.1.3 Formulas ..... 160
A.1.4 Higher-Order Logic ..... 162

## List of Figures

I. 1 Schematic diagram of the full I-DARE system ..... vii
1.1 I-DARE(lib) hierarchy ..... 3
2.1 K-Vector examples ..... 21
2.2 Replication inside blocks ..... 27
2.3 An example of wrong component tree ..... 29
3.1 Structured Instance ..... 42
4.1 Representation of $y(1-y)$ function ..... 50
4.2 a) Shared variables partitioned over substructures (except for one) in a linear model b) Variables shared by all substructures in a linear model ..... 58
4.3 Graphic relating constraints with variables for formulation §4.1.5 ..... 59
4.4 Graphic relating constraints with variables for formulation a) §4.1.5 and b) §4.1.5 ..... 61
4.5 Solver Tree assignment ..... 62
4.6 Formulation Diagram ..... 62
4.7 C++ vs. FLORA-2 hierarchies ..... 64
4.8 Overall solution process ..... 70
5.1 $\mathrm{ARR}^{\Sigma}$ reformulation and solving process ..... 86
5.2 Formulation example ..... 90
6.1 GMLS diagram ..... 100
7.1 HCP Formulation ..... 110
7.2 Independent Composition of $N$ MILP subproblems ..... 116
7.3 Composition of two MILP subproblems with shared variables ..... 118
8.1 Initial set of ARRs and structures ..... 130
8.2 Convex Structures and Reformulations ..... 136
8.3 Full structure and reformulation diagram ..... 141

## Listings

1.1 Component Class ..... 2
1.2 Leaf-Problem class ..... 3
1.3 Leaf-Problem class example ..... 4
1.4 Another Leaf-Problem class example ..... 4
1.5 Yet another Leaf-Problem class example ..... 4
1.6 MCF class ..... 5
1.7 Local Leaf-Problem class example ..... 6
1.8 Block class ..... 6
1.9 Block class example ..... 7
1.10 Another block class example ..... 7
1.11 MILP block class modification (d_loc) ..... 8
1.12 Simple Selection LfP class ..... 10
1.13 MMCF(FC) class ..... 10
1.14 Lagrangian Relax class ..... 11
2.1 Example of dimensions ..... 13
2.2 Local dimension definition ..... 13
2.3 Example of local dimensions ..... 14
2.4 Another dimension ..... 14
2.5 Class of properties ..... 14
2.6 Example of variable ..... 14
2.7 Bounding example ..... 15
2.8 Dimensions and Constants in I-DARE(im) ..... 16
2.9 Aggregator Class ..... 18
2.10 Example of aggregator general form ..... 19
2.11 Example of a specific aggregator ..... 19
2.12 LfP specification ..... 23
2.13 LfP example ..... 23
$2.14 \mathrm{LfP}^{\mathcal{L}}$ specification ..... 24
$2.15 \mathrm{LfP}^{\mathcal{L}}$ example ..... 25
2.16 Automatic property generation ..... 25
2.17 Automatic dimension and index generation ..... 25
2.18 Example of property and dimension generation ..... 25
2.19 Block definition ..... 26
2.20 LP and Integrality constraint ..... 27
2.21 MILP block instance ..... 27
2.22 Another block example ..... 28
2.23 MILP MPS LfP ${ }^{\mathcal{L}}$ class ..... 28
2.24 Formulation class ..... 29
2.25 Full formulation example ..... 30
3.1 Instance Wrapper ..... 33
3.2 XML global data format ..... 34
3.3 DH Class ..... 36
3.4 Data Handler subscription predicate ..... 36
3.5 XML Data Handler Class ..... 37
3.6 IW example ..... 37
3.7 MPS Data Handler Class ..... 37
3.8 LEAF Node general form ..... 43
3.9 BLOCK Node general Form ..... 46
4.1 Solver Root Class ..... 63
4.2 Blueprint exported to the $\mathcal{F}$ LORA-2 file ..... 64
4.3 Example of solution hierarchy ..... 65
4.4 Extension of :: $\mathcal{F}$ LORA-2 operator ..... 67
4.5 Example of Integer parameter type ..... 67
4.6 Example of Double parameter type ..... 67
4.7 Example of Integer parameter type ..... 68
4.8 Example of Choice parameter type ..... 68
5.1 ARR class definition ..... 71
5.2 $\mathrm{ARR}^{\sum}$ class definition ..... 75
5.3 $\mathrm{ARR}^{\Sigma}$ to reformulate MCF to LP ..... 81
5.4 MMCF FC class ..... 82
5.5 $\mathrm{ARR}^{\Sigma}$ to reformulate MMCF FC to Lagrangian Flow Relaxation ..... 83
5.6 Writer class ..... 85
5.7 Delegation solver $\mathcal{F}$ LORA-2 class ..... 86
5.8 Generate the path tree ..... 88
5.9 $\mathrm{ARR}^{\mathcal{A}}$ class definition ..... 88
6.1 Solver Wrapper Interface ..... 96
6.2 Machine Learning Interface ..... 97
8.1 Continuous KnapSack Problem ..... 124
8.2 MILP to block MILP ..... 124
8.3 Branch and Bound block ..... 125
8.4 MILP to Branch and bound with LP ..... 125
8.5 Semi-Continuous MILP ..... 126
8.6 MILP-SC to MILP ..... 126
8.7 MMCF(FC) to MILP-SC ..... 127
8.8 Independent Replication Selection block ..... 129
8.9 MMCF(FC) to Lagrangian Knapsack Relaxation ..... 129
8.10 Quadratic Function ..... 130
8.11 Second Order Cone Program ..... 131
8.12 Mixed-Integer Second Order Cone Program ..... 131
8.13 MISOCP to Branch and bound with SOCP ..... 132
8.14 MIQP-SC-SEP to MISOCP ..... 134
8.15 Mixed-Integer Semi-Infinite Perspective Cuts (for the quadratic case) plus Linear Constraints ..... 135
8.16 MIQP-SC-SEP to MISIPC_Q_LC ..... 136
8.17 KnapSack Problem ..... 136
8.18 Quadratic Minimum Cost Flow ..... 137
8.19 QMCF to MCF through linearization ..... 138
8.20 QMMCF FC to Lagrangian Flow Relaxation ..... 138
8.21 QMMCF FC to Lagrangian Knapsack Relaxation ..... 139

## Introduction

The development of mathematical models of reality, which often take the form of decision or optimization problems, is arguably the single most important way in which humanity improves its understanding and control over the physical world. Coupled with the phenomenal growth of available computational resources over the last 50 years, it has very substantially contributed to the exponential growth of knowledge in almost all scientific fields such as physics [109], statistics [81], data mining [86, 61], mathematics [105, 25], artificial intelligence [99, 44], and many others. Furthermore, countless many practical applications fundamentally hinge upon mathematical models in such diverse fields as transportation [29, 21], location [85, 107], scheduling [30, 90, 132], complex industrial systems [120], networks [28], bio-informatics [108], chemical engineering [23, 119], medical equipment configuration [116], and many others. Therefore, it is fundamental for the continuous improvement of science and technology that better and better mathematical models, and the software packages required to solve them, be available to researchers of all fields.

However, it is one of the most striking and important discoveries of science and mathematics in the Twentieth Century that just being able of creating a model does not mean that there is a reasonable way to solve it. Göedel's theorems [153], existence of non-computable functions [151], complexity theory [106], etc, have shown that being able to write a model is not equivalent to be able to solve it, i.e. for some models there may not be a way to compute a solution or doing it in a reasonable amount of time.

This impossibility was not clear, as testified by Hilbert in his Program [63]. Hilbert proposed to define a formulation of mathematics based on a solid and complete logical fundantion, believing that this could be done by (1) showing that all mathematics follows from a correctly chosen finite axiom system; (2) and that such axiom system is provably consistent through means such as epsilon calculus [126]. It was Hilbert's understanding that, once realized this, every possible problem in mathematics and science could have been solved by "just computing". This dream was put to end initially by Göedel, who proved that any non-contradictory formal system, which was comprehensive enough to include at least arithmetic, cannot demonstrate its completeness by way of its own axioms (any effectively generated theory capable of expressing elementary arithmetic cannot be both consistent and complete).

Göedel's results were later complemented by studies in computer theory (Turing [151], Von Neumann [155]), showing the existence of undecidable problems and non-computable functions, which of course make a model not solvable. Further down the same line, the existence of effectively computable functions, for which, can be shown that any algorithm that computes them will be very inefficient, in the sense that the running time increases exponentially (or even superexponentially) with the length of the input, which also makes the model not solvable (in a reasonable time) was discovered. Furthermore, currently, a huge number of decision and optimization problems (NPHard, NP-Complete) are believed to be unsolvable in polynomial time, although no formal proof is yet available (see [118, 100]).

Undeterred by the impossibility of a universal solution procedure (efficient enough), the scientific community has continued building better and better models. Since the "super-solver" is not available, each problem needs to be addressed individually, focusing in exploiting particularities
in their models. These particularities can be called structures, that are well known parts of a model for which exist particular solving techniques. Typically, problems can only be solved with algorithms that recognize and exploit these structures.

## I. 1 Modeling and solving with structures - Motivations

When a model of a practical industrial/scientific application is built, oftentimes a choice is made a priori (and possibly unintentionally) about which structure of the model is the most prominent from the algorithmic viewpoint. This is done by choosing first which of the several main classes of models the problem is molded in: a Linear Program (LP) [57], a Mixed Integer Linear Program (MILP) [127], and so on. This decision is mostly driven by the previous expertise of the modeler, by the "bag of tricks" she has available, and by her understanding (or lack thereof) of the intricate relationships between the choices made during the modeling phase and the effectiveness/availability of the corresponding solution procedures.

Unfortunately, making "the best" choice is arguably difficult. Many classes of models have been devised which are useful for expressing different practical problems, and the continuous improvements of solution methods have created an enormous wealth of results about different algorithmic approaches for (old and new) model classes and the conditions under which any approach is more or less computationally effective. For instance, Conic and Semidefinite Programs [31] allow for carefully selected forms of nonlinearities which keep the problems convex, and therefore efficiently solvable by appropriate classes of algorithms; they have many applications e.g. in engineering and computational mathematics, as well as having been the foundation of Robust Optimization where uncertainty of problem's data is taken into account [32].

To further enlarge the set of representable functions while still guaranteeing efficient resolvability, Disciplined Convex Programming [82, 41] require problems to be specified by following a rigorous set of rules which ensure convexity, as well as providing solution algorithms with the data they need to effectively tackle the problem. While nonconvex problems are in general much harder to solve, an enormous number of variants arise according to the specific properties of the objective function and constraints that can be exploited for algorithmic purposes. Considerable attention has recently been devoted to nonconvex nonlinear problems, with [39, 88, 144] or without [117] integrality constraints on the variables.

Several special cases of particular interest arise when nonlinearities and/or nonconvexities are a consequence of expressing specific situations, such as: the optimization of a different objective function by a different decisor $[24,68]$ or, more in general, the equilibrium between a set of different decisors [104]; constraints about joint probability of uncertain events to occur [43]; convex constraints with just one single concave component [95, 37], differential equation constraints [110, 131], and many others. Complex combinatorial structures, e.g. like the ones appearing in scheduling problems [90], can be embedded as "primitives of the modeling language" in Constraint Logic Programing (CLP) techniques [121, 22, 34] under the form specialized domain propagation techniques. And the list goes on and on.

When a model class has (more or less arbitrarily) been selected and the model has been written, a specific numerical solver has to be used to actually solve it. The typical choice is to rely on battle-hardened general-purpose solvers, capable of tackling (in principle) any one problem in the given model class without much intervention from the end-user. Unfortunately, general-purpose solvers may exhibit poor performance on many applications since they typically ignore any existing underlying structure. For each of the above problem classes, an enormous literature is available about techniques that are effective for solving specific sub-classes of problems; these include (to name just a few) preconditioning techniques in linear algebra [51, 74, 70], effective domain reduction techniques [64] and hybrid search/optimization methods in CLP [65, 93], specialized rowand column-generation algorithms in MILP [30, 33, 59, 69, 77, 92], specialized search strategies in heuristic approaches [87, 44, 73, 85], appropriate selection and breeding procedures in evolutionary programs [80], and effective learning rules in swarm-intelligence approaches [60]. Thus,
for countless applications, specialized solvers exist that are better than general-purpose ones. Yet, because they are specific to smaller classes of problems, they are often much less developed, and therefore less robust and user-friendly, than general-purpose ones. In addition, their efficiency may crucially depend on the appropriate setting of some algorithmic parameters which requires a level of understanding of their inner workings that cannot be reasonably expected outside a small circle of specialists. On top of all this, practical problems most often exhibit several structures simultaneously; not only it is not a fortiori clear which of them is computationally more relevant, but also the most efficient approach may require exploiting them all. This would call for integrating several different specialized approaches, a task most often bordering the impossible in the current state of affairs.

Thus, the fact that the most appropriate (specialized) approach is actually selected crucially depends on the realization of a long list of conditions: the user has to discover the structures (which requires knowing about them in the first place), realize that they are computationally relevant, fetch specialized numerical solvers capable of exploiting them, write the model fighting with the rigidities and quirks of the interface of the specialized solvers (such as requiring specific programming languages, using badly conceived input data formats, not allowing certain operations required by the applications, ...) and the inevitable configuration problems, and integrate all this in the environment required by the application. Often, modelers lack both knowledge and resources to perform these complex tasks; therefore the wealth of available knowledge about specialized algorithms for specific structures lies unused gathering dust in the uncharted backwaters of the scientific literature and/or in prototypical software codes which, despite holding great promises, are too specialized to be known and used outside a small circle of interested specialists. Meanwhile, end-users cannot solve their problem efficiently enough.

To make matters worse, computationally exploiting "the right" form of structure crucially depends on having chosen "the right" formulation that reveals it. However, many structures are typically not "naturally" present in the mathematical models, and must be purposely created by such weird tricks of the trade such as creating apparently unnecessary copies of variables and/or relations [84], replacing an exact compact nonlinear formulation with a much larger approximate linear one [142], replacing a single integer-valued variable by a set of binary-valued ones [69], and many similar others. That is, one needs a reformulation of the problem-which may well turn out to be rather different from the "natural" one familiar to the original modeler-where structures inside the model are transformed into other equivalent ones that are better suited to some carefully selected algorithmic approach. Finding these reformulations, and the corresponding algorithms with their appropriate configurations, is a costly and painstaking process, up to now firmly in the hands of very specialized experts-most often themselves blissfully unaware of the many potential practical applications of the techniques they master-with little to no support from modeling tools.

## I. 2 The proposal and further motivations

This Thesis will be focused in the proposal of a system, named I-DARE (Intelligence-Driven Automatic Reformulation Engine), that defines the methodology necessary to deal with the problematics issued in the previous section: structured modeling, ((re)formulation, solver, configuration) selection, structured solver application.

From the foundational viewpoint, the main aim of I-DARE is to challenge the implicit assumption, underlying all mathematical modeling efforts, that devising an effective mathematical model can only be achieved by human creativity, and that computer tools have no role on it. There are, of course, sound theoretical and practical reasons to believe that human creativity will ever-or at least for a very long time - be a necessary component of any mathematical modeling exercise. However, as for countless many other human activities before, the intervention of automated system has a huge potential to improve the efficiency and effectiveness of these efforts, ultimately allowing the human skills to concentrate on these parts that are still firmly out of reach of computer systems.

Attaining this result is, however, far from simple. First and foremost, it is a striking discovery
that while the term reformulation is ubiquitous in mathematics (e.g. [39, 82, 101, 117, 142, 144, 154] among the countless many others: a Google search on the term returns more than 600,000 hits), there are precious few formal definitions and theoretical characterizations of the concept. Among the few ones, some are limited to syntactic reformulations, i.e., those that can be obtained by application of algebraic rewriting rules to the elements of a given model [113]. These reformulations are capable of exploiting syntactical structure of the model, such as presence of particular algebraic terms in parts of its algebraic description [71, 117]. While being very relevant, these do not include all transformations that have shown to be of practical use.

Oftentimes, reformulations are based on nontrivial theorems which link the properties of two seemingly very different structures; some notable examples are the equivalent representations of a polyhedron in terms of extreme points and faces (which underpins a number of important approaches such as decomposition methods, and has many relevant special cases such as the path formulation and the arc formulation of flows [18]) and the equivalence between the optimal solution value of a convex problem and that of its dual (which is the basis of many results in robust optimization). These reformulations require a higher view of the concept of structure of a model, i.e., a semantic structure which considers the mathematical properties of the entire represented mathematical objects as opposed to these of small parts of their algebraic description; we therefore refer to them as semantic reformulations. Proper definitions of reformulation capable of capturing this concept are thin on the ground.

For instance, an attempt was made in [143] by demanding that a bijection exists between the feasible regions of the two models and that one objective function is obtained by applying a monotonic univariate function to the other, which are extremely strict conditions. A view based on complexity theory was proposed in [24], but since it requires a polynomial time mapping between the problems it already cuts off a number of well-known reformulation techniques where the mapping is pseudo-polynomial [69] or even exponential in theory [30, 59, 73], but quite effective in practice. Only recently a wider attempt at formalizing the definition of formulation has been done which covers several techniques such as reformulation based on the preservation of the optimality information, changes of variables, narrowing, approximation and relaxation [113, 114].

However, a general formal definition of reformulation is not enough for I-DARE; the aim is to identify classes of reformulation rules for which automatic search in the formulation space is possible. In this sense, syntactic reformulations, being somewhat more limited in scope and akin to rewriting systems, may prove to have stronger properties that allow more efficient specialized search strategies. Yet, defining appropriate more general classes of semantic reformulations is also necessary in order for the system to be able to cover a large enough set of possible reformulations. This calls for an appropriate definition of "structure" that on one hand is general enough, and on the other hand allows for effective search in the reformulation space.

Another crucial requirement is the ability to predict with a sufficient degree of accuracy some performance metrics of a given solution approach (with a given set of algorithmic parameters) on a given instance of a model, without actually performing the computation. This is necessary as it will provide the "objective functions" of the search, and is clearly a very difficult task. There is a huge literature on both complexity analysis and experimental evaluation of algorithms, and the system will have to be conceived as to allow exploitation of any available result for each specific solver and model classes. However, the system will also require some general-purpose approach to cover all the cases where no useful results are known. This would call for the application of machine learning techniques to the prediction of algorithms performances, and possibly for the selection of a set of "good" algorithmic parameters, a promising avenue of research which appears to have just started to produce the first concrete results [49, 96]. Yet, all the attempts so far have focused on narrow classes of problems and approaches; what would set I-DARE far apart from any other previous attempt are the sheer scale and heterogeneity of the set of approaches that have to be addressed, as well as the extension of the approach to cover the case of different formulations for the same model. Results showing that accurate prediction is indeed possible, with existing or newly devised approaches, for such a varied set of algorithms may substantially impact the practice of parameters selection in several applications.

From the technological viewpoint, none of the currently available methodologies and tools for mathematical modeling provides all the functionalities envisioned and needed by I-DARE:

1. A modeling language capable of representing semantic structures, providing the user with a rich set of constructs that permits a representation of the problem which is "natural" form and independently from the underlying solution methods.
2. A core system capable of automatically reformulating the models and a search mechanism in the space of (formulations, algorithms, configurations) that is capable of finding "the best" formulation of a given problem, intended as the one which, applying the selected algorithm with its selected configuration, provides the most efficient solution approach.
3. A general solver interface capable of integrating specialized solution approaches by making it possible for one solver to use others as sub- or co-solvers independently from their algorithmic details; this calls for a structured instance description language which, unlike currently available ones, is flexible enough to allow passing all parts of the data of the instance to each different involved specialized solvers in the format it requires and supports.

While 2. is arguably the most innovative feature, the other two components are also crucial for the overall success of the system. Remarkably, the need for these is indeed felt in the modeling and numerical solver communities, as witnessed by the fact that they have been addressed to some extent in several existing software projects. Yet, each of these project has focussed on specific aspects of the problem, without addressing the whole (challenging) general issue.

Regarding need 1, standard algebraic modeling languages like AMPL [66] or GAMS [15] are completely "unstructured": they offer no support to partitioning a model into sub-components with clearly defined interfaces that may be developed and modified separately from each other. While you can define subproblems and have each of them solved by a different solver, each submodel lives in the same namespace, and changes in any of them may (unintentionally) bear changes in the others. Thus, first-generation algebraic languages can be likened to the first generation of computer languages like FORTRAN or Basic in this respect, and share the same weaknesses: developing and maintaining large and complex models, while possible, gets rapidly extremely difficult as the size increases. The need for more structured modeling languages is clearly felt in the community, and it has been addressed in several ways. One is to rely on an existing Object-Oriented Programming (OOP) language, in order to inherit its structured programming capabilities: this is for instance the case of FLOPC++ [4] (using C++), puLP [134] (using Python) and of the commercial OptimJ product [130] (using Java). While this may help, it does not introduce "natural" constructs specifically for modeling; besides, it requires knowledge of the host programming language. Similarly, CP-based approaches like G12 [76] "naturally" allow for some degree of structured modeling due to the partial extensibility of the CP language, but they ultimately remain tied to a specific modeling and solving paradigm - although hybrid CP/MILP approaches are possible in SCIP [16] and ILOG (now IBM) Concert [54]-and to the underlying programming language ( C or $\mathrm{C}++$ ). The need for providing structure directly at the algebraic modeling level is addressed e.g. by the RIMA project [136], and is especially felt in the context of what is often referred to as multidisciplinary design or multidisciplinary optimization, giving rise to projects like ASCEND [156] and pyMDO [122] (both using the OOP capabilities of Python). However, the aim of these projects is "only" to make it simpler for the user to come up with a correct model; once that is done, the formulation is "flattened up" and passed to a general-purpose solver.

The main proposal of I-DARE in this respect is to move up a further step of the ladder of expressive power, devising a modeling system based on declarative languages like Prolog [129]. In particular, Frame-Logic systems [102] like FLORA-2 [159] allow to combine the expressive power of declarative languages with OOP components, providing tools of unparalleled effectiveness for reasoning about structures. This is clearly necessary to deal with semantic (non-syntactical) structures, since then reformulations are not limited to application of algebraic rewriting rules, but require the capability of checking logical conditions for the applicability of a given reformulation rule.

Since logic programming paradigms, while extremely powerful, are even less familiar to the vast majority of perspective users than OOP ones, this logic-based modeling language should not be the only choice for interfacing with the system proposed in this Thesis. Instead, its primary role will be that of an intermediate modeling language, used to drive the core search and reformulation capabilities of the system (cf. 2. above) but largely invisible both to end-users and to algorithms developers. For the former, a number of different existing front-end systems may be adapted to produce the required intermediate language representation of the model; this is for instance the case of ASCEND (which comes already equipped with a nice GUI component) or pyMDO, but also extensions to popular alternatives like GAMS and AMPL can be considered, a-la SML [53]. Back-end communication with algorithms will be obtained through a general solver interface and a structured instance format, discussed next.

In order to work as planned, the system I-DARE, requires a unified interface for "every possible solver". The primary component of such an interface will have to be a(n extensible) structured instance format. File formats for solvers are often awkward remnants of the punched-cards era such as MPS [97], rather difficult to understand for all but the most technically-savvy users [140], and/or extremely fragmented so that the same model can be represented in several different incompatible ways [125] to suit the needs of the different available solvers. This makes it more difficult to collect instances of models for testing and validation purposes, requiring substantial work for the trivial and uninspiring task of converting one (awkward) data format into another. A unified data format with good expressive capabilities, thanks to the flexibility of XML, has been recently proposed in the Optimization Services project [137], extending the concept to that of data stream, e.g. served by a network connection. While the OS format may provide a convenient starting point, it will be necessary to extend it to explicit support for semantic structures, where some parts of the model are represented in "abstract" terms by providing indication of the intended semantic of the model rather than some algebraic description of its constraints. A logic and very convenient consequence of this choice is that the instance format should not be intended to replace every existing specialized data format, but rather to be a meta-format which, other than allowing to "natively" represent the data and the algebraic structures of the instance, may delegate the task of representing specific instance blocks (corresponding to specialized structures) to specialized data formats. This has several advantages, apart from that of not wasting resources re-inventing the wheel (for instance, the native format itself may be delegated to Optimization Services). It will be possible to use existing solvers with their natively supported data formats, without the need for an interface layer for a new language. Existing instances sets will not need to be translated in the new format, which may be error-prone and may increase their size (the flexibility of XML being often dearly paid in terms of size bloat) w.r.t. that of possibly highly compact specialized encoding. The data format will evolve together with, and adapt to, the set of supported solvers. Finally, the reformulation machinery of the core system will yield a universal data format translator, capable of automatically perform the conversion between any two sets of specialized formats for any two classes of models for which a (chain of) reformulation rule(s) exists in the system.

The universal data format will serve as basic input structure for a universal solver interface. The need for isolating the user from the details and quirks of the underlying solution methods is heavily felt, as demonstrated e.g. by the Open Solver Interface project [123] and the Stochastic Modeling Interface [12], both in COIN-OR [1], and by the MCFClass project [124]. Yet, these are limited to very specific classes of solvers and models. The traditional approach to the solver independence problem has been that of delegating the interface to the (algebraic) modeling language such as GAMS or AMPL; however, this is only possible when the underlying solver is a monolithic, generalpurpose one.

The aim of the Thesis in this matter is to allow different solvers for specialized problems to be used together; therefore, a mechanism for generic solver collaboration will have to be devised and perfected. It is well-known that a trade-off between generality, efficiency and flexibility exists so that a mechanism devised for covering all potential uses is unlikely to be possible, or at least efficient enough. As for the structured instance format, the idea is therefore that of exploiting the semantic information embedded in the structured model description to allow solvers for a
given structure to impose constraints on the set of functionalities provided by other solvers they collaborate with, thus rendering the universal solver interface flexible and capable of incorporating existing interfaces like the previously mentioned ones.

## I. 3 I-DARE - Overview

Figure I. 1 provides a schematic description of the I-DARE system. The system is divided into three main parts: front-end, core system and solving section, clearly separated by interfaces.


Figure I.1: Schematic diagram of the full I-DARE system

The front-end part will comprise one or more graphical and/or textual front-ends for the system. A general Modeling Environment Handler (ME-Handler) must be designed that declares all functionalities that I-DARE exposes to each modeling environment, effectively setting the interface between the front-end and the core system. The interface will rely on I-DARE(im), the I-DARE logic-based intermediate modeling language. Defining the ME-Handler, will not be considered as a part of this Thesis, whereas a detailed definition of I-DARE(im) will be provided.

The core system is further subdivided in three parts: formulation and reformulation, performance evaluation, and control.

- The formulation part (denoted as "Structures and ARRs" in Figure I.1) is responsible for the definition of the structured model and of the corresponding structured instance, together with the set of reformulation rules that can be used to perform reformulations. It is based on I-DARE(im), and composed by three modules. I-DARE(lib) is the package containing all the structures that I-DARE knows, together with the basic mechanisms to reason about formulations, such as verifying their well-formedness. When a formulation is complemented with a set of actual data it becomes an Extended Model (instance in standard optimization parlance); a general Instance Handler (I-Handler) will be defined, which allows data to be retrieved from the different sources (files, databases, network connections, ...) using newly defined and/or existing data formats (cf. I-DARE(ei)). A general deduction system, denoted as I-DARE( t ) will then be used to reformulate EMs by using the database $A R R$ of Atomic

Reformulation Rules, together with the necessary argument (input) and answer (output) mappings between the two concerned structures. While most ARR could conceivably be directly applied to models, they are in general applied to EM for two reasons. The first is that applicability (or exact form) of a reformulation may depend on the actual data of an instance. The second is that gauging the computational impact of a reformulation cannot typically be done without some access to the actual data; this is in fact the rationale for the next component.

- The performance evaluation part (denoted as "ML" in Figure I.1) is responsible for the delicate task of predicting the performances of each tentative reformulation, so as to compute the "objective function" which guides the search (itself governed by the next component). While it is clear that the performance evaluation will require some form of machine learning, it is fundamental to decouple the basic structure of the system from the details of the specific learning method employed. Therefore, the ML approach will be "seen" by the control mechanism as a "black box", described by an abstract General Machine Learning Control $G M L C$ interface. Internally, a similar interface (denoted as $\Psi$ in the figure) will be defined to allow any general ML approach to be used to actually perform the prediction. Some notable details of the system, evidenced in the figure, require further comments. First, since a "solver" may actually be composed by a combination of several different solvers, each one will possibly have a different ML approach (denoted by $\Psi_{S}$ in the figure). Second, given the generality of the system, prediction will necessarily have to be solver-specific, with each solver at least extracting its tailored set of "features" from the instance and presenting them to the ML; this is the task of solver wrappers, that will have to be implemented to hook any numerical solver to the system. Third, while any solver will be able to rely on the "external" general-purpose ML approach, some solvers may have the capability of self-predicting their performances, either using highly tailored ML approaches or completely different techniques like complexity analysis. This "internal" ML will have to be appropriately presented to the general mechanism, e.g. to be used to compute the performance of the "overall" solver from these of its "sub-solvers". Other crucial components of this part of the I-DARE system are the actual continuous Learning mechanism which - either exploiting actual runs by end-users or using spare CPU cycles to perform test runs - updates the knowledge base upon which the prediction is performed, and the Meta Learning mechanism that allows to evaluate and compare different ML techniques for computing the $\Psi$ function, thereby selecting those which provide the best results.
- Finally, the control part is composed by the package I-DARE(control), in charge of guiding the search for the best (reformulation, solver, configuration) $(f, s, c)$ triplet. In order to decouple the basic structure of the system from the details of the search mechanism-the appropriate choice of which will require substantial research - the package defines the abstract interface that any control mechanism will have available to guide the deduction process of the I-DARE $(\mathrm{t})$ package for generating the tentative reformulations, whose performances will be predicted by the $G M L C$ component, until the desired $(f, s, c)$ triplet is reached.

Finally, the solving section is responsible for actually performing the solution approach on the chosen (re)formulation, collecting and presenting the result to the core system, which will in turn refactor them in the format of the original instance to present them to the front-end. Its interface with the core system is the I-DARE's Enhanced Instance format I-DARE(ei), itself composed by the three ( $f, s, c$ ) parts:

- SInstance is the actual encoding of the final instance, obtained at the end of the reformulation process, represented in the structured instance format;
- Solver Tree is the description of the (set of) numerical solver(s) that has(have) been selected by the search process as the most appropriate for the given SInstance; as previously mentioned, this is not just a solver but, in general, a structured collection of solvers, some of which using others to cope with specialized structures;
- Configuration is the description of the configuration(s) that has(have) been selected by the search process as the most appropriate for (each solver in) the Solver Tree and the given SInstance.

The I-DARE(solve) package will then have to orchestrate the actual solution process, possibly taking into account issues like distributed computation, relying on the available set of Solver Handlers ( $S$-Handler), i.e., implementations of the general solver interface that allows to plug specific solvers to the I-DARE system.

## I. 4 Thesis structure

This Thesis is be divided in an Introduction, eight Chapters and the Conclusions. This subdivision will provide the reader with a definition step by step of all modules in I-DARE. Chapter 1, describes how structure classes and their relations can be created and stored, by defining the I-DARE(lib) module. Chapter 2 defines the internal modeling language I-DARE(im), formally describing all concepts of well-formedness, from the dimensions and indexes, to leaf problems, blocks and formulations. Chapter 3, specifies how the data can be attached to the formulations, by using the proper format handlers and wrappers; concluding with the definition of the structured instance. Chapter 4, describes how solvers can be attached to I-DARE, and how they can be configured and linked to the structured instance, by means of a Solvers' Tree. In chapter 5 we define all the necessary theory related with reformulations; we describe the concept of Atomic Reformulation Rules (ARR), how I-DARE will treat them semantically, and how they can be applied to a formulation to define the Reformulations' Domain. Chapter 6 describes the I-DARE(control) module; it will describe the available search spaces and how can they be used to search for the "best" triple (formulation, solver, configuration). In Chapter 7 we declare a set of "simple" structure classes' examples that will allow (by composition) the creation of complex models, and using these classes we will define some reformulations in order to obtain a MILP (or LP) structure. Whereas, in Chapter 8 we will focus on a set of structures for which there are specialized solution methods, and we will study how we can relate them by applying the appropriate reformulation rules. Finally the conclusions will sum up the results obtained and will describe the potentials and future research.

## Chapter 1

## I-DARE(lib) - the structure library


#### Abstract

$\qquad$ I-DARE allows the construction of models based on an extensible structure class library. This structure class library contains a set of basic components that enables the definition of new classes of structures and how these classes will interact between each other. Components of I-DARE(lib) will be included inside a hierarchy, abstracting the main characteristics of each structure class. Furthermore, this hierarchy enables the user, by adding new pieces, to enlarge I-DARE(lib)'s potential. This chapter defines I-DARE(lib) in a bottom-up fashion, starting from the most basic components like dimensions and parameters types, advancing to atomic problems and ending with problems compositions (blocks). At the end of this chapter some detailed examples are presented, together with a discussion about the usage of declarative programming (in particular FLORA-2 - cf. Appendix A) to define I-DARE(lib).


### 1.1 Basic components

Every structure class is ultimately reduced to a set of parameters (plus the semantical meaning of the structure class). The parameters represent the characteristics of the input and output of the structure class. For example, let's examine a LP structure class:

$$
\begin{aligned}
& \qquad \min / \max \sum_{i} c_{i} v_{i} \\
& \text { s.t. } \\
& \qquad \sum_{i} c_{j, i}^{\prime} v_{i} \leq /=/ \geq b_{j} \quad \text { for all } j
\end{aligned}
$$

To create an "instance" of the LP structure class, we need to specify the direction (min or max), the vector in the objective function, the matrix in the constraints, the right hand side vector, the relation that will be used in each constraint and the vector of variables on which the solution will be stored. Moreover, we need to know how the cardinalities of the previous elements are related to each other. Observe that the number of objective function constants must correspond to the number of variables and the number columns in the constraint matrix. These cardinalities will be called dimensions (see $\S 2.1$ ). For instance the previous LP structure class has two dimensions, that can be called columns and rows.

Definition 1.1.1 (Dimension Meta Variable (dMV)) To define how many dimensions a structure class will have, I-DARE(lib) will use the property dim_var->[ $\left.d_{1}, \ldots, d_{k}\right]$, where $k$ is the number of dimensions, and $d_{i}$ and $d_{j}$ are identifiers, such that $d_{i} \neq d_{j}, \forall i, j \in[1 . . k]$. Each $d_{i}$ will be called dimension Meta Variable (dMV) and it represents a set $\left[0 . .\left\|d_{i}\right\|-1\right] .\left\|d_{i}\right\|$ states for the dimension cardinality (formally defined in 3.1.1).

For the LP structure class the dMV list may be, dim_var $->$ [cols, rows].

### 1.1.1 Parameter Types

As mentioned previously, parameters are a key element in the definition of a structure class, therefore when defining a structure class in I-DARE(lib) we must be able to specify the types of potential parameters. These potential parameter types can be,

- d_var - Variable type
- d_constant - Constant type
- d_vector(?K,?S) - Vector type
- d_rel - Relation type
- d_direction - Direction type


## Variable and Constant Types (d_var and d_constant)

These two types are used to denote the variables and constants of the structure class respectively. For instance, in the previous LP structure class example, $x_{i}$ is of variable type and $c_{i}$ is of constant type. Formal definition on well-formed variables and constants will be given in $\S 2.3$.

## Vector Type (d_vector (? $\mathrm{K}, ~ ? \mathrm{~S})$ )

When we use dimensions while defining constants, variables, relations or even general expressions a vector may be produced. The vector type is denoted using d_vector(?K, ?S), where ?K $\in\{$ d_var, d_constant, d_rel \} and ?S must be a non-empty list of dMVs. |?s| represents the number of dimensions of the vector (e.g. a matrix is a 2 -dimensional vector).

For instance, d_vector(d_constant, [cols]) represents a one dimensional vector of constants. On the other hand, when $\{$ d_vector(d_constant, [cols ]), d_vector(d_var, [cols]) $\}$ appear in the same parameter list (of a certain structure), it indicates that the number of elements of the first and second vector are the same. Well-formed vectors will be formally defined in $\S 2.3$.

## Relation Type (d_rel)

Another kind of argument is the relation, denoted using d_rel. Relations allowed by I-DARE will be binary over $\mathbb{R} \times \mathbb{R}$, like, $=$ (equality),$=<$ (less than or equal) and $>=$ (grater than or equal).

## Direction Type (d_direction)

There are structure class that may contain an objective function within them. In this case, the structure class may require the specification of objective function's direction as one of its parameters. Supported directions are min and max. Although both directions are transformable multiplying the objective function by -1 , we decided to keep them both for the sake of expressibility.

### 1.2 Structure Classes

I-DARE(lib) defines a basic hierarchy for the structure classes that one may create. This hierarchy is composed of three classes, d_Component_C, d_LeafProblem_C and d_Block_C (see figure 1.1).

The most general structure class (d_Component_c) will define the most general behaviors that every other structure class must implement. This class is defined as follows,

Listing 1.1: Component Class

```
d_Component_C[
    abstract,
    // Methods and Properties
```



Figure 1.1: I-DARE(lib) hierarchy

```
    => wellformed,
    freeinds => _list,
    allinds => _list,
    pureinds => _list,
    vars => _list
].
```

Methods and Properties

- wellformed - tells whether a component is well-formed or not.
- vars - retrieves the list of variables used within a component.
- freeinds - retrieves the list of free indices of a component.
- pureinds - retrieves the list of all indices used as constants within expressions in the component.
- allinds - retrieves the list of all indices used within the component.

In the declaration of d_Component_C appears a class boolean field, named abstract. When this field appears in a class definition the system will not allow the creation of instances from that class.

The definition of the d_Component_C class introduces for the first time the concept of index. Indices will be formally defined in $\S 2.2 .3$, but let's give at least an empirical definition that may be useful in the rest of the chapter.

An index is an identifier linked to a dimension (and used to iterate over that dimension). For instance, $x_{i}$ from the LP example, uses the index $i$ that iterates over dimension cols. The free indices of a component in I-DARE are the set of indices that are not fixed by any construct like vectors or cumulative operators like $\sum$. For example if we use the constant $c_{i j}^{\prime}$ (with indices $i, j$ ) which appears inside $\sum_{i} c_{i j}^{\prime}$ then $j$ would be a possible free index (if not fixed elsewhere).

All methods in d_Component_C are inherited and/or overwritten by all its descendant classes.

### 1.2.1 Leaf Problem Class

As a special case of component, I-DARE(lib) defines the class of leaf problems. A leaf problem is an atomic definition composed of at least an objective function or a constraint. For example, linear problems, disjunctive constraints and quadratic objective functions are leaf problems. Henceforth leaf problems will be called LfP.

All LfPs must inherit from the following class,
Listing 1.2: Leaf-Problem class

```
d_LeafProblem_C :: d_Component_C
[
    abstract,
    [local]
    // Methods and Properties
    dim_var => _list ,
    [dim_bound => _list ,]
    args => -list
].
```


## Methods and Properties

- dim_var - defines the list of dMVs of the LfP.
- dim_bound - specifies which constants will have as domain a dMV's set of values. optional
- args - defines the parameter types that the LfP will require.

The list of arguments args must be defined as a dictionary [name $=$ parameter type ...], where name is an identifier unique inside the LfP.

A simple example of LfP would be the class of integer problems,
Listing 1.3: Leaf-Problem class example

```
d_IC_C :: d_LeafProblem_C
[
    dim_var -> [d],
    args -> [
        ivar = d_vector(d_var, [d]) // variables to be integer
            ]
].
```

This class will define the problems having a set of integer variables (ie. ivar ${ }_{i} \in \mathbb{Z}, i \in \mathrm{~d}$ ). As another example, one can define the class of linear constraints with its list of argument types.

Listing 1.4: Another Leaf-Problem class example

```
d_Linear_Constraints_C :: d_LeafProblem_C
[
    dim_var -> [d1,d2],
    args -> [
        x = d_vector(d_var, [d1]), [d2,d1]), /// variables
        b = d_vector(d_constant, [d2]), // b vector
        rels = d_vector(d_rel, [d2]) // relations for each constraint
        ]
].
```

The LfP d_Linear_Constraints_C represents all constraints with the form $\sum_{i} A_{j, i} x_{i}$ rel $_{j} b_{j} \forall(j)$. We may also define the class of linear problems (LP),

## Listing 1.5: Yet another Leaf-Problem class example

```
d_LP_C :: d_LeafProblem_C
[
    dim_var -> [cols, cons],
    args -> [
        x = d_vector(d_var, [cols]), // variables
            c = d_vector(d_constant, [cols]), // price constants
            A = d_vector(d_constant, [cons, cols]), // A matrix
            b = d_vector(d_constant, [cons]), // b vector
            rels = d_vector(d_rel, [cons]), // relations for each constraint
            dir = d_direction // objective function direction
            ]
].
```

Note that in this case we are defining a complete LP with objective function and constraints. Once we use the parameter type d_direction we are giving a hint that within that component there must be an objective function. This leaf problem is represented algebraically in the following way,

$$
\begin{aligned}
& \text { dir } \sum_{i} c_{i} x_{i} \\
& \text { s.t. } \\
& \qquad \sum_{i} A_{j, i} x_{i} \text { rel }_{j} b_{j} \quad \forall(j)
\end{aligned}
$$

Looking at LfP's properties we find one that has not been previously defined, the dim_bound. This property declares which constant parameter (vector or not) must have a domain restricted by a dMV. The dim_bound list must have the following form: [(CN,D),...], where args [CN] = d_constant or d_vector(d_constant, ?_), and Dedim_var. This property essentially constraints the parameter CN to take values between 0 and $\|\mathrm{D}\|-1$.

The next example will be a Minimum Cost Flow LfP (MCF). For this LfP we used the classical graph representation and we wrote it into I-DARE(lib) syntax. This example will illustrate the usage of dim_bound

Listing 1.6: MCF class

```
d_MCF_C :: d_LeafProblem_C
[
    dim_var -> [N,E],
    args -> [
        SN = d_vector(d_constant, [E]), // start nodes
        EN = d_vector(d_constant, [E]), // end nodes
        SD = d_vector(d_constant, [N]), // supply/demand
        cost = d_vector(d_constant, [E]), // cost per arc
        u = d_vector(d_constant, [E]), // arc capacity
        flow = d_vector(d_var, [E]) // flow variables
            ],
    dim_bound -> [(SN,N), (EN,N)]
].
```

We used two dMVs, one to represent the nodes ( N ) and the other to represent the arcs ( E ). Using those dMVs we created a set of parameters, to represent:

- the arcs with SN and EN , so each arc will be $\left\langle\mathrm{SN}_{i}, \mathrm{EN}_{i}\right\rangle, i \in[0 .|\mathrm{E}|-1]$;
- the supply/demand of each node, supply implies a positive value, demand a negative one, otherwise must be 0 (SD);
- the the cost and the capacity of each arc (cost and u);
- the flow variables (output of the structure) (flow).

Note that the SN and EN parameters must have values between 0 and $\|\mathbb{N}\|-1$; in fact they are restricted by the dim_bound property.

A MCF may be also represented using a LP. If we define $N^{+}(i)=\{j \mid(i, j) \in \mathrm{E}\}$ and $N^{-}(i)=$ $\{j \mid(j, i) \in \mathrm{E}\}$, representing the outgoing and incoming arcs, respectively, then

$$
\begin{array}{ll}
\min & \sum_{(i, j) \in \mathrm{E}} \operatorname{cost}_{i j} \mathrm{flow}_{i j} \\
\text { s.t. } \\
& \sum_{j \in N^{+}(i)} \text { flow }_{i j}-\sum_{j \in N^{-( }(i)} \text { flow }_{i j}=\mathrm{SD}_{i} \quad i \in \mathrm{~N} \\
\quad 0 \leq \operatorname{flow}_{i j} \leq \mathrm{u}_{i j} \quad(i, j) \in \mathrm{E}
\end{array}
$$

Using this LP formulation we may apply any LP solution approach to solve it. However, explicitly recognizing MCF as a structure class, allow us to apply specific solution approaches, like:

- Cycle Canceling: a general primal method [103];
- Minimum Mean Cycle Canceling: a simple strongly polynomial algorithm [78].
- Successive Shortest Path and Capacity Scaling: dual methods, which can be viewed as the generalizations of the Ford-Fulkerson algorithm [62].
- Cost Scaling: a primal-dual approach, which can be viewed as the generalization of the push-relabel algorithm [79].
- Network Simplex: a specialized version of the linear programming simplex method [124].

These algorithms have proved to be more efficient for several types of graphs, with respect to the plain LP approach.

## Local Leaf Problem Class

In the LfP class definition we specified an optional property local. When this property is present it indicates that the potential instances will define all their data in a particular local format (e.g. MPS, OSiL, DIMACS, etc). A LfP class that contains the property local will be called Local Leaf Problem ( $\mathrm{LfP}^{\mathcal{L}}$ ) Class. An example of $\mathrm{LfP}^{\mathcal{L}}$ class may be,

Listing 1.7: Local Leaf-Problem class example

```
d_LP_MPS_C :: d_LeafProblem_C
[
    dim_var -> [cols, cons],
    args -> [
            x = d_vector(d_var, [cols]), // variables
            c = d_vector(d_constant, [cols]), // price constants
            A = d_vector(d_constant, [cons,cols]), // A matrix
            b = d_vector(d_constant, [cons]), // b matrix
            rels = d_vector(d_rel, [cons]), // relations for each constraint
            dir = d_direction // objective function direction
            ],
    local
].
```

d_LP_MPS_C represents a class of LPs that takes all data from MPS format files. The parameter types in the case of $\operatorname{LfP}{ }^{\mathcal{L}}$ classes represent what data the $\operatorname{LfP}^{\mathcal{L}}$ will export to be used globally. At least variables should always be present in the parameter type list, otherwise there will be no communication between the $\operatorname{LfP}^{\mathcal{L}}$ and the rest of the model. In any case, it is always advisable to include the maximum possible set of parameter types, due to problem reformulation requirements (see §5).

### 1.2.2 Block Class

Problems can be built from the composition of other subproblems. For instance, e MILP problem can be seen as the composition of a LP and integer constraints. These compositions, will be called, blocks. Every block must inherit from the following class.

Listing 1.8: Block class

```
d_Block_C::d_Component_C [
    abstract,
    //Methods
    ids => -list,
    subsC => _list,
    link => _list,
    [rp|R => -list]
].
```


## Methods

- ids - represents a list of identifiers, one for each substructure class.
- subsC - is the list of substructure classes.
- link - is a list that contains how the variables of the substructure classes must be related,
- rpIR - is an optional field that represents a list of elements of the form ?id $=$ ? dt, where ?id $\in$ ids and ?dt must be an algebraic expression involving the following operands,
- a term $A(d)$, where $A \in$ ids and $d \in A . d i m \_v a r ; ~ o r ~$
- a template item (c.f. Definition 1.2.6).

The substructure classes will represent the structures that are grouped by this block, and the link represents how the variables inside those structures will interact with each other (see §1.2.3 for formal definitions).

Here is an example to illustrate how a block could be constructed:
Listing 1.9: Block class example

```
d_B_MILP_C :: d_Block_C
[
```



```
    link ->> [[X,Y], [X] ]
].
```

In this case we are defining the MILP class using a block construction. The first sub-structure is a LP class and the second one is an Integrality Constraint (IC) class. But how do we ensure that the set of variables to be integer is a subset of the variables in the LP sub-component? For doing that we use the link list.

A link is a template of the variables to be used in the substructure of a block. Each member of a link correspond to a substructure. For example, $[\mathrm{X}, \mathrm{Y}]$ correspond to d_LP_C and $[\mathrm{X}]$ to d_IC_C. In this case $[X, Y]$ is telling us that the variables of its corresponding substructure have to be exported in two groups, the variables linked to $X$ and the variables linked to $Y$. But since there is a second element in link, $[\mathrm{X}]$, we need to ensure that the variables exported by d_IC_C and the first group of d_LP_C are the same.

There is a formal definition for link's behavior, that can be seen in $\S 1.2 .3$. The most important part of this definition is the Different Name Unification rule DNU, that ensures that the link unifies with the exported variables satisfying that equal templates must correspond to identical variable groups and different templates must correspond to variables groups with no element in common. The following example will illustrate this fact.

Listing 1.10: Another block class example

```
d_B_vardept_C :: d_Block_C
[
    ids -> [master , slave ],
    subsC }->\mathrm{ [d_Component_C, d_Component_C],
    link > [[X,V] , [X,W]
].
```

In this case the first and second substructure class must export an identical group of variables for X but must export groups of variables for V and W with no elements in common. In section $\S 2.5$, we will see more detailed examples of how the variables can be exported ensuring the DNU rule.

### 1.2.3 Block - Formal definitions

This section will introduce step by step all the elements that compose a block, defining them formally.

Definition 1.2.1 (Class item) Given ?C:: d_Component_C, then a class item can be defined as one of the following two terms:

1. ?C - represents one structure of class ?C for which the system will try to assign a solver,
2. d_loc (?C) - represents one structure of class ?C for which the system will not try to assign a solver

When the system is trying to assign a solver to a block, and it encounters a class item of the form d_loc(?C), it will completely ignore that substructure. The system will assume that the solver assigned to the container block will deal with that substructure.

For instance we could modify the d_B_MILP_C example indicating that the d_IC_C structure will be treated inside the solver assigned to d_B_MILP_C, using the d_loc modifier.

Listing 1.11: MILP block class modification (d_loc)

```
d_B_MILP_C :: d_Block_C
[
    ids - [lp, ic (loc(d_IC_C)],
    link ->[[X,Y], [X] ]
].
```

Using the previous definition we can define the subsC property of blocks, as a list of class items. On the other hand, the ids property must be a list of non-repeated identifier, such that $\mid$ ids $|=|$ subsc $\mid$.

While integrating several substructures together shared variables may occur. The following four definitions will focus on how variables can be exported from substructures.

Definition 1.2.2 (Variable Tuple) Let $V$ be a set of variables, then a variable tuple of $V$ is $\left(v_{1}, \ldots, v_{m}\right)$, where $0<m \leq|V|, v_{i} \in V, i \in[1 . . m], v_{i} \neq v_{j}, i \neq j \in[1 . . m]$. If $m=1$ the parenthesis could be removed. The empty tuple will be represented by ().

For example if $V=\{v, w, x, y\}$ then the following are variable tuples:

- (),
- $w$,
- $(v, y)$,
- $(y, v)$ and
- $(w, y, x)$.

On the other hand, $(w, y, w)$ is not a variable tuple, because the $w$ is repeated. A permutation of a variable tuple generates a different one. For example, $(v, y)$ and $(y, v)$ are different variable tuples.

Definition 1.2.3 (Disjoint Variable Tuples) Let $v t$ and $w t$ be two variable tuples of sets $V$ and $W$, respectively, then $v t$ and $w t$ are disjoint if they do not contain common variables.

For instance, $(v, x)$ and $(w, y)$ are disjoint, whereas $(v, y)$ and $(w, y, x)$ are not, because they have $y$ in common.

Then finally using variable tuples we will be able to construct lists of tuples that denote how the block will recognize the variables of its sub-structures.

Definition 1.2.4 (Variable Pattern) Let $V$ be a set of variables, then a pattern over $V$, is a list $L$ of variable tuples of $V$, such that $\bigcup_{i}^{|L|} L_{i} \subseteq V$ and $L_{i} \cap L_{j}=\emptyset$ for all $i \neq j \in[1 . .|L|]$.

For example, the following are variable patterns:

- $[v,(x, y)]$,
- $[(w, x),(y, v)]$ and
- $[v, y, w]$

Also a permutation of the variable tuples inside a variable pattern generates a different one. For instance, $[v, y, w]$ and $[y, v, w]$ are different variable patterns.

Definition 1.2.5 (Disjoint Variable Patterns) Let $v p$ and $w p$ be two variable patterns of sets $V$ and $W$, respectively, then $v p$ and $w p$ are disjoint iff $\forall(v t \in v p) \forall(w t \in w p)[v t$ and $w t$ are disjoint].

For instance, if $W=v, w, r, t$ then $[v,(x, y)]$ and $[(r, t), w]$ are disjoint, whereas $[v,(x, y)]$ and $[(r, t),(v, w)]$ are not disjoint, because $v$ and $(v, w)$ are not disjoint.

Block classes when using the list link define how the variable patterns of its potential substructure objects must be arranged. Next definitions are focused on that matter.

Definition 1.2.6 (Template Item) A template item will simply be an atomic identifier.
For example X , var are template items; and $\mathrm{p}(\mathrm{q})$ is not a template item.
Definition 1.2.7 (Template pattern) Given TI a list of template items with no repeated elements, and the atom d_all (called modifier) then a template pattern is defined as,

1. TI or
2. (TI, d_all).

For instance,

- $[\mathrm{X}, \mathrm{Y}]$,
- ( $[\mathrm{X}, \mathrm{Z}]$, d_all $)$.
are template patterns. On the other hand, $[\mathrm{X}, \mathrm{X}]$ is not a template pattern since X is repeated inside the template pattern. The d_all semantics will be describe later in this section while defining unification.

Definition 1.2.8 (Variable Tuple - Template Item unification) Let $v t$ be a variable tuple, $t i$ be a template item, then we say that $v t$ and $t i$ always unify.

Definition 1.2 .9 (Template pattern - Variable Pattern unification) Let pt be a template pattern and $v p$ be a variable pattern constructed over the set of variables $V$, then pt unifies with $v p$ iff

1. if $p t$ has not the form ( $s p t$, d_all) then $|p t|=|v p|$,
2. if $p t$ has the form ( $s p t, \mathrm{~d}_{-}$all) then $|s p t|=|v p|$ and $v p$ has to contain all the variables in $V$.

For example ([X,Y,Z], d_all) unifies with $[(v, w), y, x] \in V$, since $\mathrm{X}=(v, w), \mathrm{Y}=y$ and $\mathrm{Z}=x$ and the variable pattern uses all variables in $V$.

Using the definition of template pattern, we may define the link property of a block class as a list of template pattern, such that $\mid$ link $|=|$ subs $C \mid$.

As a next step we will define the different name unification rule (DNU). This rule defines a change in the classical unification extremely necessary for block specification (see $\S 2.5$ ).

Definition 1.2.10 (Different Name Unification rule) Let $L$ be a link composed of $t p_{i}$ template patterns, and $V P L$ be a variable pattern list composed of $v p_{i}$, then we say that $L$ and $V P L$ ensure the Different Name Unification (DNU) rule iff

1. $|L|=|V P L|$ and
2. $t p_{i}$ unifies with $v p_{i}$ and
3. Let $t$ be a template item of $t p_{i}$ (without the modifiers) and $V_{t}$ be the corresponding (unified) variable tuple in $v p_{i}$, then $\forall(t) \forall\left(t^{\prime}\right)\left[t=t^{\prime} \Rightarrow V_{t}=V_{t^{\prime}}\right]$ and $\forall(t) \forall\left(t^{\prime}\right)\left[t \neq t^{\prime} \Rightarrow V_{t} \cap V_{t^{\prime}}=\emptyset\right]$ (i.e. the same template items in link must be unified with the same variable tuples and all the different template items with disjoint variable tuples).

For instance if the link of a block class is $[[\mathrm{X}, \mathrm{Y}],[\mathrm{X}, \mathrm{Z}]]$ and the variable pattern list of an object created from that block class is $[[(x, y), z],[(x, y), w]]$, then both of them ensure the DNU, being $\mathrm{X}=(x, y), \mathrm{Y}=z$ and $\mathrm{Z}=w$. But if the variable pattern were $[[(x, y), z],[(x, y),(w, z)]]$ then there would be a violation of the DNU. In this case the unification is $\mathrm{Y}=z$ and $\mathrm{Z}=(w, z)$ and see how $z$ is repeated in both $Y$ and $Z$, which are different template patterns.

Finally we would like to talk about the functionality of the field rpIR. This field represents what we call replication relations, defining for the selected substructures the exact number of replications that can be done, based on the cardinality of the dMV or the number of variables unified with a template item. A substructure will be replicated depending on its free indices (see §2.5), hence a real verification of the replication relations could only be done when we have knowledge of the actual data (see §3.1.3).

### 1.3 Some Structure Class examples

In the first example we will define a LfP class for a really simple problem, but that will become useful later on in the thesis. This LfP will be call Simple Selection and its algebraic representation is the following,

$$
\begin{aligned}
& \qquad \operatorname{dir} \sum_{i} f_{i} y_{i} \\
& \text { s.t. } \\
& \qquad y_{i} \in\{0,1\}
\end{aligned}
$$

As can be seen this problem requires for its solution a mere inspection of the cost $f_{i}$ to decide which $y_{i}$ will have value 0 or 1 in order to minimize or maximize the function. The I-Dare(lib) structure class representing the Simple Selection LfP will be,

Listing 1.12: Simple Selection LfP class

```
//** The variables (y) will be considered binary {0,1}.
d_Simple_Selection_C :: d_LeafProblem_C
[
dim_var -> [d1],
args -> [
        dir = d_direction,
        y = d_vector(d_var, [d1]), // binary variables
        f = d_vector(d_constant, [d1]) // constants
        ]
].
```

Note that even if the basic numeric type in I-DARE is $\mathbb{R}$, a structure class may impose new constraints to that basic type, expressed in the structure class' semantics. A similar situation was already presented (d_IC_C) whereby the variables were constraint to be of integer type, and still there are no explicit syntactical construct to express this basic type constraint. This is because the structure classes define the parameter types just syntactically (i.e. the input/output of the structure), while the rest lies in the semantical value of the structure, adequately documented. This semantical information serves as a guideline to the potential solvers, that may eventually be "registered" to that structure.

The next example, will be a Multi-commodity Minimum Cost Flow with Fixed-Charge LfP (MMCF(FC)). This problem's representation is very similar to the MCF LfP class (see Listing 1.6). It includes one new dimension K (commodities); the costs will now depend also on the commodities; there are fixed costs per arc; the capacities are divided into single and mutual; and there is a new set of variable, the design variables (boolean variables that determine which design arc is being used).

```
//** Note that K represent the commodities and N the nodes
d_MMCF_FC_C :: d_LeafProblem_C
[
    dim_var }->[N,E,K]
    args -> [
        SN = d_vector(d_constant, [E]), // start node
            EN = d_vector(d_constant, [E]), // end node
            c = d_vector(d_constant, [K,E]), // cost per unit per arc
            f = d_vector(d_constant, [E]), // fixed cost per design arc
            SD = d_vector(d_constant, [K,N]), // supLfPy/demand
            b = d_vector(d_constant, [K,E]), // single k capacity per arc
            u = d_vector(d_constant, [E]), // mutual arc capacity
            flow = d_vector(d_var, [K,E]), // flow variables
            desg = d_vector(d_var, [E]) // design variables
            ],
    dim_bound }->>[(SN,N), (EN,N)
].
```

The $\operatorname{MMCF}(\mathrm{FC})$ LfP class uses the dim_bound property to constraint SN and EN to $[0 . .\|\mathrm{N}\|-1]$. Using the same notation applied for MCF, we can derive an arc-based MILP representation of the MMCF (FC) class,

$$
\begin{aligned}
& \min \sum_{k \in K} \sum_{(i, j) \in E} \mathrm{c}_{i j}^{k} * \mathrm{flow}_{i j}^{k}+\sum_{(i, j) \in E} \mathrm{f}_{i j} \operatorname{desg}_{i j} \\
& \text { s.t. } \\
& \sum_{j \in N^{+}(i)} \mathrm{flow}_{i j}^{k}-\sum_{j \in N^{-}(i)} \text { flow }_{j i}^{k}=S D_{i}^{k} \quad i \in N, k \in K \\
& 0 \leq \text { flow }_{i j}^{k} \leq \mathrm{b}_{i j}^{k} \quad(i, j) \in E, k \in K \\
& \\
& \sum_{k \in K} \operatorname{flow}_{i j}^{k} \leq \mathrm{u}_{i j} \quad(i, j) \in E \\
& \sum_{k \in K} \mathrm{flow}_{i j}^{k} \leq \mathrm{u}_{i j} \text { desg }_{i j} \quad(i, j) \in E \\
& \text { flow }_{i j}^{k} \leq \mathrm{b}_{i j}^{k} \operatorname{desg}_{i j} \quad(i, j) \in E, k \in K \\
& 0 \leq \operatorname{desg}_{i j} \leq 1 \quad(i, j) \in E \\
& \operatorname{desg}_{i j} \text { integer } \quad(i, j) \in E
\end{aligned}
$$

Using the MILP formulation we could apply simplex-based cutting plane methods [27] that benefit from the wide availability of codes for solving LP. However, these methods generally do not exploit any underlying structure and the LPs may become huge. So, again in this case, the fact of recognizing the $\mathrm{MMCF}(\mathrm{FC})$ structure explicitly allow us to apply focused solution methods like, Lagrangian relaxations (flow, knapsack relaxations, etc) [69] and heuristics [55].

Finally, we will present a block class. This class will represent blocks that can be solved using Lagrangian Relaxation Methods. This Lagrangian Relaxation structure will split the problem into a substructure plus linear constraints.

Listing 1.14: Lagrangian Relax class

```
d_B_Lagrangian_Relax_C :: d_Block_C [
    ids > [ sub, linking],
    subsC -> [ d_LR_C, d_loc(d_Linear_Constraints_C)],
    link > [([X],d_all), ([X],d_all)]
].
d_LR_C :: d_Component_C [abstract]. // Auxiliary structure class
```

To create this block class, we used an auxiliary class (d_LR_C) that inherits directly from component and its abstract, this class has an explicit semantical meaning strictly related with the interface of its potential solvers (this topic will be treated in §4.4).

The d_B_Lagrangian_Relax_C block, has a substructure class and a linking structure class. Note that the linking structure class (d_Linear_Constraints_C) has the modifier d_loc, indicating that there will be no solver assignation for that substructure (i.e. the solver assigned to d_B_Lagrangian_Relax_C must deal with it). In fact, this structure is used to construct the Lagrangian dual, by updating the objective function information in the substructures (sub) [111].

### 1.4 Discussion

I-DARE(lib) is an extensible library and the base of the modeling section of I-DARE. Extending the I-DARE(lib) hierarchy we can define new problem classes and/or new combination of problems. Each non-abstract class defined in I-DARE(lib) represents an specific structure for which there must exist solution techniques or at least reformulation rules (see $\S 5$ ).

The definition of all components inside I-DARE(lib) using declarative programming ( $\mathcal{F}$ LORA-2) permits us to take advantage of the powerful deduction engine of $\mathcal{F}$ LORA- 2 and the expressiveness of its language. At the same time we use the object-oriented characteristics of $\mathcal{F}$ LORA- 2 while defining I-DARE(lib) hierarchy.

Another cause that moved us to use declarative programming is that defining and implementing the system becomes almost one task. Also the fact that we can change the code being executed without stopping the system and apply changes dynamically, is another useful feature of $\mathcal{F}$ LORA-2 (generally present in Logic Programming).

I-DARE(lib) is not a package that contains computationally heavy processes, in fact its main purpose is to store all the structures and to define simple verification mechanisms to ensure wellformedness of the classes.

To use $\mathcal{F}$ LORA-2 in I-DARE(lib)'s definition gives us future query potentiality. Since I-DARE(lib) is essentially a "database" of structures, querying it will be crucial for future processes, like reformulating or solving. Hence using $\mathcal{F}$ LORA-2 to define I-DARE(lib) enables us to use Frame-Logic + HiLog + Transactional-Logic power to consult the structure class database.

## Chapter 2

## I-DARE(im) - base modeling environment


#### Abstract

$\qquad$ Using the components defined in I-DARE(lib) the user has at her disposal the structure base to create models. For that purpose we define I-DARE(im), an internal modeling environment based on I-DARE(lib) and also designed in $\mathcal{F L O R A}$-2.

First we will see the main constructs that can be used, which will permit us to represent dimensions, constants, variables, vectors and others. Furthermore, we will present their wellformedness rules, until we arrive to a complete definition of a formulation, representing the internal model (IM) of the problem. It is called internal model because above this modeling environment we could create modeling languages (maybe graphical) that represent the models in a more user friendly manner. I-DARE(im) was chosen as base modeling environment because it gives us a formal and sound way to create a formulation, and thus its design is based on $\mathcal{F L O R A}$-2 its implementation becomes more natural.

To define I-DARE(im) a bottom-up fashion will be used, starting from the dimension and properties and ending with the leaf-problems, blocks and formulations.


### 2.1 Dimensions

Variables and constants may have one or more dimensions. A dimension is a finite set that defines a way of indexing variables, constants, leaf problems and also blocks.

A dimension in I-DARE(im) is represented by the fact d_dimension(?id) where ?id is a unique atom. We could define two dimensions as an example, the first referring to production plants and the second one to products,

Listing 2.1: Example of dimensions

```
d_dimension(plant).
d_dimension(product).
```


### 2.1.1 Local dimensions

Dimensions can be also automatically defined by the system from $\operatorname{LfP}^{\mathcal{L}} \mathrm{S}$ (see $\S 2.4 .1$ ). In this case we will add an extension to the d_dimension/1 predicate (previously defined), a d_dimension/2 predicate.

Listing 2.2: Local dimension definition
1 d_dimension (? $\mathrm{X}, \quad$ ?structure).

This predicate, like the previous one, specifies an atom representing the name of the dimension (?x). As second argument it requires a component's name (?structure), representing the $\operatorname{LfP}^{\mathcal{L}}$ that defines the dimension.

For example, assume C is a $\operatorname{LfP}^{\mathcal{L}}$ that specifies the usage of a certain dimension D . Then the system will generate the following dimension,

Listing 2.3: Example of local dimensions
1 d_dimension (D, C).
But now assume the user defined a dimension,
Listing 2.4: Another dimension
1 d_dimension (D).
This dimension defined by the user will be called global. When a global and a local dimensions have the same name, both of them must represent the same set of elements and therefore have the same cardinality.

### 2.2 Variables and Constants

Variables and constants can be seen as properties present in the problem being modeled. However, we need to make a distinction between variables and constants because they will be treated differently while solving a problem. We will construct a common root class and using d_var, d_constant and multi-inheritance, the mentioned distinction will be enabled.

The common root will be called d_property and it is defined as follows,
Listing 2.5: Class of properties

```
d_property
[
    dims => _list ,
    lower => _double,
    upper => _double,
    => bounded
].
```

where

- dims - is the list of all dimension of the property,
- lower - is a lower bound to the property,
- upper - is an upper bound to the property,
- bounded - defines whether it is a bounded property or not.

A variable must be defined as an object which name is an atom and it is an instance of d_property and d_var. For instance,

Listing 2.6: Example of variable

```
x:d_var
x:d_property
[
    dims -> [d1, d2]
].
```

defines an unbounded variable named $x$, which has two dimensions d1 and d2.
For constants the same principle is applied, but d_constant is used instead.

### 2.2.1 Bounding

Defining the bounds of a property may be done using lower and/or upper. One may specify numerical values, however I-DARE(im) allows the usage of constants to define bounds. One may create a bounded or partially bounded property $p$ in two possible ways:

1. Using a number or
2. Using a constant $c$ ( $c$ :d_constant).

The first case is the simplest one, we should only place a number in the definition of the bound and it is done. For instance, the following code,

Listing 2.7: Bounding example

```
x:d_var.
x:d_property [
    dims -> [d1, d2],
    lower m 0,
    upper -> 1
].
```

defines a variable that has to take values greater or equal than 0 and lower or equal than 1.
The usage of constants is syntactically simple, but one may incur in some inconsistencies related with the properties' dimensions. For instance, if we take property $\times$ an try to lower bound it with a constant c that has dimensions d 3 that $\times$ does not have. In this case, when we try to obtain the lower bound of $\mathrm{x}_{i, j}(i \in \mathrm{~d} 1$ and $j \in \mathrm{~d} 2)$ we wont know which c to use, because we have no idea which value must be assigned to dimension d3.

Since a property c may be bounded by a constant c' that may be as well bounded by another constant c", and so on, we need a way to synthesize this sequence of bounding applications. We can thereby analyze what happens when we have cycles in those sequences.

Definition 2.2.1 (Bounding spanned sequence (BSS)) Let $a_{1}$ be a constant and $a_{1} . b\left(a_{2}\right)$ be the fact that $a_{2}$ is a bound (lower or upper) of $a_{1}$, assuming we have the following sequence $a_{1} . b\left(a_{2} . b\left(a_{3} . b\left(\ldots . . a_{n}\right)\right)\right.$, where $a_{n}$ is an unbounded constant or bounded by a number, then the sequence $a_{1}, \ldots, a_{n}$ will be a bounding spanned sequence ( BSS ) of $a_{1}$.

Proposition 2.2.2 Let $S$ be a $B S S$ of a property $p$, then $S$ is finite if there are no repeated elements inside $S$.

The proof of this proposition is quite straightforward, assume $S$ is the BSS of a constant c and that c is repeated inside $S$, then $S$ must recursively contain the BSSs of the repeated c, to which the same principle applies. Therefore $S$ is not finite.

Next definition summarizes the bounding conditions exposed above,
Definition 2.2.3 (Bounding Rule) Let $p$ be a property and $c$ a constant, if using $c$ for bounding $p$ (second case) we have to ensure the following

- If $D_{p}$ are the dimensions of the property and $D_{c}$ are the dimensions of the constant, then $D_{c} \subseteq D_{p}$.
- All bounding spanned sequences of $p$ must not contain repeated elements. (no-cycle condition)


### 2.2.2 Property well-formedness

In this section we will formally define the well-formedness rules of properties in general, and its particular cases: variables and constants.

Definition 2.2.4 (Property well-formedness) We say $p$ is a well-formed property iff

1. $p$ :d_property, being $p$ a unique atom,
2. $p$ [dims $->$ ? L ], where ? L is an empty list, or a list composed of well-formed dimensions,
3. bounding in $p$ satisfy Rule 2.2.3 (Bounding Rule).

Definition 2.2.5 (Variable well-formedness) We say $v$ is a well-formed variable iff

1. $v$ is a well-formed property and
2. $v:$ d_var.

Definition 2.2.6 (Constant well-formedness) We say $c$ is a well-formed constant iff

1. $c$ is a well-formed property and
2. $c:$ d_constant.

### 2.2.3 Indexing

When accessing a property, indexing is required. Indexing will allow the future definition of vectors (of variables or constants).

An index is an atom that it is associated to a dimension. It is declared using the fact d_index(?id, ?dim), where ?id is an atom and ?dim is a defined dimension (i.e. d_dimension(?dim)). There could be more than one index defined over one dimension. An index is well-formed iff ?id is unique and ?dim is a well-formed dimension.

Let prop:d_property be a property with dimensions $\left[d_{1}, \ldots, d_{n}\right](n>0)$, then we call the term $\operatorname{prop}\left(i_{1}, \ldots, i_{n}\right)$ an indexed property, where d_index $\left(i_{1}, d_{1}\right), \ldots, \boldsymbol{d}_{-} \operatorname{index}\left(i_{n}, d_{n}\right)$. If $n=0$ then the indexed property must just be the term prop.

For instance, assume we have the following piece of I-DARE(im) model,
Listing 2.8: Dimensions and Constants in I-DARE(im)

```
d_dimension(product).
d_dimension(plant).
d_index(i, product).
d_index(j, plant).
stock:d_constant.
stock:d_property
[
    dims -> [product, plant]
].
```

One possible indexed property of stock could be stock $(i, j)$.
When indexing we may also consider the possibility of using several indices over a single dimension. For instance consider the property x with a single dimension D , we could declare an index i over $D$ and use the indexed property $\times(i)$. However, we could use more that one index over a single dimension. For example, assume we have $j$ over D1 and $k$ over D2, then we could use the indexed property $\times((\mathrm{j}, \mathrm{k}))$. Note that we grouped j and k using a tuple, this will imply that the actual index that is going to used is the linear combination $j *$ _card (D1) +k (operator _card will be explained in the following section). The only condition we must satisfy is that the cardinality of the tuple must be equal to the cardinality of the corresponding dimension.

An indexed property is well-formed iff the property is well-formed and the indices used are also well-formed. We will also use the concept of well-formed indexed variable and well-formed indexed constant, which definitions are straightforward from the previous ones.

### 2.2.4 Expressions

Once we have indexing we can define how expressions in I-DARE(im) can be built. Expressions will use a set of operators defined by the predicate d_operator(?op, ?c), where ?op is the operator symbol and ?c is the arity of the operator. An initial set of operators will be included in I-DARE(im) like, $+,-, *, /,^{\wedge}$ and the special operators _card, _up and $\_^{\prime} w\left({ }^{\wedge}\right.$ is the power operator, _card applied to a dimension retrieves its cardinality, _up applied to a property returns its upper bound and _lw applied to a property returns its lower bound). There are also 0 -ary operators, called system constants, like _pi ( $\pi$ constant), _e ( $e$ constant), _big (arbitrarily large positive number). If a property $\times$ is asked for its upper/lower bound and the bound is not defined then the operator $\operatorname{up}(x) /\lrcorner \mathrm{lw}(\mathrm{x})$ will return _big/-1*_big.

The priority of the basic operators is set as usual. The _card, _up and _lw operators do not need priority because they will never be applied in infix form. Expressions in I-DARE(im) will take the following operands:

- numbers,
- indices,
- direction,
- indexed properties and
- aggregators (see 2.2.4).

For example, a well-formed expression could be $1+c(i) * x(i, j)$, where $c(i)$ is an indexed constant and $\times(i, j)$ is an indexed variable.

An indexed property inside an expression produces a replication of the expression while generating the instance. This replication is done for each value taken by the indices of the indexed property. For instance if i can take values over $\{1,2\}$ and j over $\{1,2,3\}$ then the previous expression is replicated in the following way

$$
\begin{aligned}
& 1+c(1) * x(1,1) \\
& 1+c(1) * x(1,2) \\
& 1+c(1) * x(1,3) \\
& 1+c(2) * x(2,3) \\
& 1+c(2) * x(2,3) \\
& 1+c(2) * x(2,3)
\end{aligned}
$$

As can be seen indices i and j in the previous expression are free indices. If they were fixed inside the expression the outcome might be different, as will be seen in further examples.

When we use a direction as an operand the system will assume that min evaluates to 1 and that max evaluates to -1 . This permits, for instance, to dynamically unify the direction of two linear objective functions. See that if we multiply the constants by the direction we are unifying to a minimization function. If we use $-1 *$ direction instead, then we are unifying to maximization function. This will become very useful in the definition of some reformulations.

We will have a particular type of expression, called constant expression. A constant expression can not contain any reference to an indexed variable. For instance the previous expression is not a constant expression.

## Conditions

While constructing a model we may need to use logical expressions. In I-DARE(im) this logical expressions are called conditions. A condition uses as operators the following fixed set:

- = - equality,
- $=\backslash=-$ not equality,
- <- less than,
- > - greater than,
- = $<$ - less or equal than,
- $>=$ - greater or equal than,
- , - conjunction,
- ; - disjunction and
- not - negation.

This set of operator is used to construct conditional expressions that will help us build the models; there is no relation between this set of logical operators and the one represented by d_rel. The last one is used as a parameter type to structure classes, in order to represent inequations or equations (the constraints in the structure).

Condition's operands will be the atom true and constant expressions. For example c(i,j)< _card(plant) is a valid condition which involves an indexed property $(c(i, j)$ and the cardinality operator applied to a dimension (plant)).

Conditions' well-formedness is exactly extracted from $\mathcal{F}$ LORA-2's predicates definition, just adding the concept of constant expression's well-formedness.

Using conditions I-DARE(im) enables the construction of case sequences. A case sequence will define several choices of constant expressions, more or less like a switch-case operator in languages like C++.

To build a case sequence we will use the _cs(?S) operator, ?S being a non-empty list equal to $\left[? \mathrm{CE}_{1}->?_{1}, \ldots, \mathrm{CE}_{k}->? \mathrm{C}_{k}, ? \mathrm{CE}_{k+1}\right.$ ], where

- ? $\mathrm{CE}_{i}$ - is a constant expression or a d_rel (see $\S 1.1 .1$ ),
- ? $C_{i}-$ is a well-formed condition.

For example, $[c(i)->(i>3, i<20), c(i) * 5->(i>=20), 0]$, represents a case sequence with two choices, if the index $i \in(3 . .20)$ it takes the value of constant $c(i)$, if $i>=20$ takes the value of constant expression $c(i) * 5$, and if $i<=3$ takes the value 0 .

Note that case sequences require a final element that has no condition attached, this element may be called the default element. A case sequence will be evaluated from left to right sequentially, and it will stop at the first condition that turns out to be true. In case no condition is true, the default is returned.

## Aggregators

In I-DARE(im) we have a special type of operator that aggregates replicated expressions. This operator is called aggregator.

Aggregators are operators that deal with the free indices of an expression, iterating on some of these indices. All aggregators must inherit from the class d_aggregator. The class d_aggregator defines the signature of a boolean method, called wellformed. The semantics of wellformed changes from one aggregator to another, as will be seen later.

Listing 2.9: Aggregator Class

```
d_aggregator
[
    => wellformed,
    = cons_wellformed,
    = exempt(?I),
    freeinds => -list,
```

```
    allinds => _list,
    pureinds => -list ,
    vars => _list
    eval(?IW, ?FI) -> ?res
] .
```

See that other methods must be defined

1. cond_wellformed - Must verify whether the aggregator is well-formed and has no variables in it,
2. exempt(?!) - Given an index list, it must verify whether the aggregator and all within it does not fix any of the ?।, but just for the variables,
3. freeinds - Must retrieve the free indices of the aggregator,
4. allinds - Must retrieve all the indices (free or not) used inside the aggregator,
5. pureinds - Must retrieve all the indices used as operands inside the aggregator (called pure indices),
6. vars - Must retrieve all the variable names used inside the aggregator,
7. eval - Defines the semantics of the aggregator, it receives a wrapper to the instance data ?IW (see $\S 3.1$ ), and the values of the indices that has already been fixed ?FI. It returns in ?res the evaluation of the aggregator.

When creating an aggregator, we may do it in the following way,
Listing 2.10: Example of aggregator general form

```
?id(?expr, ?inds, ?cond):d_aggregator.
?id(?expr, ?inds, ?cond) [ wellformed ] :- ...
.... // implement all methods
```

where

- ?id, is a unique atom representing the name of the aggregator,
- ?expr, is the expression to aggregate,
- ?inds, is the list of indices that will be used to aggregate,
- ?cond, is a condition.

This is really a suggestion of how we may define an aggregator. At the end one can define the aggregator however one likes, just ensuring the inheritance and the method implementations.
For example, lets see the aggregator a_sum
Listing 2.11: Example of a specific aggregator

```
a_sum(? expr, ?inds, ?cond):d_aggregator.
a_sum(?E, ?I, ?C)[wellformed] :-
    expr_wellformed(?E), //verifies ?E is a well-formed expression
    expr_freeinds(?E, ?el), //retrieves free indices of ?E (?el)
    subset(?el, ?l, ?_), //verifies that ?l \subseteq ?el
    cond_wellformed(?C), //verifies condition well-formedness
    cond_freeinds(?C, ?cl), //retrieves free indices of ?C (?cl)
    subset(?el, ?cl, ?_). //verifies that ?cl \subseteq ?el
        // implement all other methods
```

The aggregator identifier has to be added as an operator (d_operator (a_sum(?_? ? ? ? - ), 3)).
In the previous code example we showed the wellformed method implementation, that defines when the a_sum is well-formed or not. Observe that the last subset call enforces that the conditions must not contribute to increase the free indices of an expression.

For example $c(i) * \operatorname{assum}(x(i, j),[j], j>0)$ is an expression that has $i$ as free index, see how $j$ is in the list of indices of a_sum and at the same time is used in the condition, to iterate over the $\times(\mathrm{i}, \mathrm{j})$ with $\mathrm{j}>0$.

Other aggregators can be defined like _min(?L)/_max(?L), which return the minimum/maximum of the specified list ? L .

Once the concept of aggregator is defined, we can formalize we can formalize some concepts that we have been using so far, like free index of an expression and exempt for expressions.

Definition 2.2.7 (Free index of an expression) Let $p$ be a property that belongs to an expression $E$ and $I$ the index list of $p$, then if there is an $i \in I$ that is not contained in any aggregator index list, it is called free index of $E$.

Definition 2.2.8 (Exempt for expressions) Let $E$ be a well-formed expression and ?। a list of indices, we say $E$ is exempt(?।) iff

- $E$ is a number, an index or a constant expression;
- if $E$ is an indexed variable and $v I$ the indices in $E$ then $? \subseteq \subseteq v I$;
- if $E$ is an aggregator and $E_{1}$ is the expression within $E$ with at least one variable reference inside of it, then $I \cap$ (indices to be fixed by $E)=\emptyset$ and $E_{1}$ [exempt(? $\left.)\right]$ must be true;
- if $O$ is the most external operator in $E$ and $L O_{E}$ is the list of operands of $O$ then $\forall\left(\varepsilon \in L O_{E}\right)$ $\varepsilon$ must be exempt(?).

For example, if inside a structure $C$ we have an indexed variable $x(i, j)$ that is used inside an expression like $\sum_{i} x(i, j)$ then if we ask $C$ whether it is exempt in [ $\left.\mathrm{i}, \mathrm{j}\right]$ or not, the answer would be no, because the index $i$ is fixed and it is used inside an indexed variable $(x(i, j))$.

The exempt definition is only interested in ensuring that a certain index list is not fixed for any variable (left as free indices).

### 2.3 Scalars and Vectors

Scalars and vectors define the division between the parameters with no dimensional replication and parameters with dimensional replication, respectively. The scalars will be the set of constant expressions, variables, directions and relations.

Definition 2.3.1 (Scalar well-formedness) A scalar $s$ is well-formed

1. with respect to d_var iff $s$ is a well-formed indexed variable,
2. with respect to d_constant if $s$ is a well-formed constant expression (including case sequences),
3. with respect to d_rel if $s$ is $\mathrm{a}=,=<$ or $>=$,
4. with respect to d_direction if $s$ is $\min$ or max.

Scalars' definition does not affect the free indices of its inner components. Therefore the free indices of a scalar will simply be the free indices of the property or of the expression used to define it. In case the scalar is of type d_rel or d_direction the free index set will always be $\emptyset$.

On the other hand vectors are always represented by the type d_vector(?K, ?S) and represent the multi-dimensional collections of parameters.

A vector in I-DARE(im) is declared using two types of constructs:

1. $\$(? x$, ?inds) - where ?x is a scalar, ?inds is the list of indices or tuples of indices that will be used to build the vector.
2. $\$(? \mathrm{KV})$ - where ?KV is a K-vector (cf. Definition 2.3.5).

For example, $\$(\mathrm{a}(\mathrm{i}, \mathrm{j}),[\mathrm{j}])$, with $\mathrm{j} \in[0 . .2]$ generates the vector, $\langle a(i, 0), a(i, 1), a(i, 2)\rangle$.
In the second way of defining vectors we used a new concept called $K$-vector. A K-vector is a structure built using a series of lists placed one inside the other to obtain a sort of multi-level list, like a set of pyramids. This structure is used to concatenate multi-dimensional vectors.

For instance, assume we have the vectors $\$(\mathrm{a}(\mathrm{i}),[\mathrm{i}])$ and $\$(\mathrm{~b}(\mathrm{j})$, $[\mathrm{j}])$, we may want to obtain a final vector product of the concatenation of the previous ones. To make the problem more complex assume we want to concatenate 2 -dimensional vectors or 3-dimensional vectors. The Figure 2.1, show some graphical examples until 3 dimensions.

2-dimensional K-Vector


3-dimensional K-Vector


Figure 2.1: K-Vector examples
To formally define a K-Vector, we will first need several definitions. These definitions will guide us step by step towards the final specification of K-Vector.

Definition 2.3.2 (Multilevel List) $K V$ is called multilevel list iff

1. $K V=\left[V_{1}, \ldots, V_{m}\right]$ where $V_{i}$ has the form $\$$ (?type, ?inds), or
2. $K V=\left[K V_{1}, \ldots, K V_{m}\right]$ where $K V_{i}$ is a multilevel list.

A multilevel list is simply a collection of embedded lists using a level fashion, such that in the final level we find vectors of the form $\$$ (?type, ?inds). For instance,
$E=[[\$(\mathrm{a}(\mathrm{i}, \mathrm{j}),[\mathrm{i}, \mathrm{j}]), \$(\mathrm{~b}(\mathrm{i} 1, \mathrm{j} 1), \quad[\mathrm{i} 1, \mathrm{j} 1])], \quad[\$(\mathrm{c}(\mathrm{i} 2, \mathrm{j} 2),[\mathrm{i} 2, \mathrm{j} 2])]]$ is a multilevel list.
Definition 2.3.3 (Levels' description) Given a multilevel list $K V$, we define its levels' description, denoted levels ( $K V$ ), as folows,

1. if $K V=\left[V_{1}, \ldots, V_{m}\right]$, where $V_{i}$ has the form $\$$ (?type, ?inds), then levels $(K V)=[m]$.
2. if $K V=\left[K V_{1}, \ldots, K V_{m}\right]$, where $K V_{i}$ is a multilevel list, then levels $(K V)=\left[m \mid \operatorname{levels}\left(K V_{1}\right)\right]$.

For example, levels $([[\$(a(i, j),[i, j]), \$(b(i 1, j 1),[i 1, j 1])], \quad[\$(c(i 2, j 2),[i 2, j 2])]])=[2,2]$.
Definition 2.3.4 (Deepness rule) A multilevel list $K V$ satisfies the deepness rule iff

- $K V=\left[V_{1}, \ldots, V_{m}\right]$, where $V_{i}$ is a vector of the form $\$$ (?type, ?inds); or
- $K V=\left[K V_{1}, \ldots, K V_{m}\right]$, where $K V_{i}$ is a multilevel list, $\forall i, j \in[1 . . m]$ levels $\left(K V_{i}\right)=\operatorname{levels}\left(K V_{j}\right)$, and $K V_{i}$ satisfies the deepness rule.

Note that a multilevel list $K V$, with levels $(K V)=[\mathrm{m} \mid \mathrm{R}]$, that satisfies the deepness rule, is composed by $m_{1}$ elements with identical levels' description equal to $[\mathrm{R}]$. Moreover, |levels $(K V) \mid$ denotes the deepness of the multilevel list. For instance, the previously defined $E$ does not satisfy the deepness rule since the multilevel sublists have different levels' description [2] and [1]. An example of multilevel list that satisfy the deepness rule may be,
$E 1=[[\$(\mathrm{a}(\mathrm{i}, \mathrm{j}),[\mathrm{i}, \mathrm{j}]), \$(\mathrm{~b}(\mathrm{i} 1, \mathrm{j} 1), \quad[\mathrm{i} 1, \mathrm{j} 1])], \quad[\$(\mathrm{c}(\mathrm{i} 2, \mathrm{j} 2), \quad[\mathrm{i} 2, \mathrm{j} 2]), \$(\mathrm{~d}(\mathrm{i} 3, \mathrm{j} 3), \quad[\mathrm{i} 3, \mathrm{j} 3])]]$.
Definition 2.3.5 (K-Vector) Let:

- $K V$ be a multilevel list that satisfies the deepness rule and has levels $(K V)=l$;
- $K V\left[m_{0}\right]\left[m_{1}\right] \ldots .\left[m_{|l|-1}\right]=\$\left(?\right.$ type $_{m_{0}, \ldots, m_{|l|-1}}$, ? inds $\left._{m_{0}, \ldots, m_{|l|-1}}\right)$, with $m_{j} \in\left[0 . . l_{j}-1\right]$ and $j \in$ [0.. $|l|-1]$;
 with $\mathbf{i} \in[0 . .|l|-1]$,
then $K V$ is a $K$-Vector iff $\forall m_{j} \in\left[0 . . l_{j}-1\right], \forall j \in[0 . .|l|-1]$ and $\forall i \in\left[0 . . l_{j}-1\right]$

1. $\mid ?$ inds $_{m_{0}, \ldots, m_{|l|-1} \mid} \mid=l$; and
2. $-\operatorname{card}\left(?\right.$ inds $\left._{m_{0}, \ldots, m_{i}, \ldots, m_{|l|-1}}[|l|-\mathrm{i}]\right)={ }^{\operatorname{cord}}\left(?\right.$ inds $\left._{m_{0}, \ldots, m_{i}, \ldots, m_{|l|-1}}[|l|-\mathrm{i}]\right), \forall m_{i}^{\prime} \neq m_{i} \in\left[0 . . l_{i}-1\right]$.

For example, $E 1$ is a K-Vector if $|\mathrm{i}|=|\mathrm{i} 1|,|\mathrm{i} 2|=|\mathrm{i} 3|,|\mathrm{j}|=|\mathrm{j} 2|$ and $|\mathrm{j} 1|=|\mathrm{j} 3|$. Henceforth, we will access the vectors inside a K-vector as if it were flattened (there were no levels). This may simplify the notation a little bit.

Definition 2.3.6 (Vector well-formedness) A vector $\bar{v}$ is well-formed with respect to d_vector(?K, ?S) iff

- if $\bar{v}$ has the form $\$(? x$, ?inds) then

1. ?x must be a well-formed scalar with respect to ?K,
2. ?inds must be a list of well-formed indices or tuples of well-formed indices,
3. $\mid$ ?inds $|=|$ ?s $\mid$,

- if $\bar{v}$ has the form $\$(K V)$ then

1. $K V$ is a K-Vector and $|\operatorname{levels}(K V)|=|? \mathrm{~S}|$,
2. $\forall \bar{v}^{\prime} \in K V, \bar{v}^{\prime}$ is a well-formed vector.

In practice vectors formed using a K-vector, will also accept scalars (if they are of the correct type). Note that a scalar can easily be seen as a vector of one dimension of cardinality 1 . For instance, $\$([\$(\mathrm{a}(\mathrm{i}),[\mathrm{i}]), 1,0])$ is a well-formed vector, 1 and 0 are two scalars and they will be appended at the end of the vector $\$(a(i),[i])$. Of course scalars can not be concatenated if the K -Vector has more than one level.

Lets define the free indices of a vector.
Definition 2.3.7 Let $\bar{v}$ be a vector, if it is defined using

- $\$(? x$, ?inds), then the free indices of $\bar{v}$ will be $F-$ ?inds, where $F$ are the free indices of ? ,
- $\$(K V)$, then the free indices of $\bar{v}$ will be $\bigcup_{i \in[1 . . n]} F_{i}$, where $F_{i}$ are the free indices of $\bar{v}_{i} \in K V$.

To retrieve all indices and pure indices can be defined in an analogous way.
Lets also define when a vector is exempt(? ? ),
Definition 2.3.8 Let $\bar{v}$ be a vector, then $\bar{v}$ is exempt(?।) iff

- if $\bar{v}$ has the form $\$(? \times$, ?inds), then ? $\times$ has to be exempt(? $)$ and ?inds $\cap$ ? $1=\emptyset$.
- if $\bar{v}$ has the form $\$(K V)$, then $\bar{v}^{\prime}$ has to be exempt(?।) $\left(\forall \bar{v}^{\prime} \in K V\right)$.


### 2.4 Leaf Problem

A leaf problem (LfP) is represented in I-DARE(im) using the following object definition,
Listing 2.12: LfP specification

```
?id:?class
[
    args -> ?args,
].
```

where

- ?id is a unique atom
- ?class ::d_LeafProblem_C[not abstract, not local] is a non abstract non local LfP class,
- ?args is the dictionary of arguments of the LfP, similar to the one used in §1.2.1.

LfP well-formedness will be mainly focused on ensuring the argument's well-formedness. For instance, if the type specified in the LfP class is a scalar (d_var or d_constant) then the LfP must specify a valid scalar in the argument list, and the same if it is a vector, a relation or a direction.

Definition 2.4.1 (LfP well-formedness) A LfP $P_{k}$ is well-formed iff

1. ?id is a unique atom,
2. ?class :: d_LeafProblem_C[not abstract, not local ]),
3. $\mid$ ? args $|=|$ ?class . args $\mid$,
4. for all (?id = ?a) $\in$ ? args,
(a) if ?class ? ? args [? id ] $\in\left\{d \_\right.$var, d_constant, d_rel $\}$then ?a must be a well-formed scalar with respect to ?class ? ? args[?id];
(b) if ?class ? ? args[? id ]=d_vector(?K, ?S) then ?a must be a well-formed vector with respect to ?class .? args[?id];
(c) if ?class .? args[? id $]=$ d_direction then $? \mathrm{a} \in\{\min , \max \}$.

The following LfP is an object instance of the LP class (see §1.2.1),
Listing 2.13: LfP example

```
ex1 : d_LP_C
[
    args -> [
        x = $(x(i),[i]),
            c = $(1, [i]),
            A =$(a(j,i), [i,j]),
            b = $(_cs([0-> (j<1), 1]), [j]),
            rels =$('=<', [j]),
            dir = 'min'
        ]
].
```

Listing 2.13 can be represented algebraically as follows,

$$
\begin{aligned}
& \min \sum_{i} 1 * x_{i} \\
& \text { s.t. } \\
& \qquad \sum_{i} A_{j, i} x_{i} \leq\left\{\begin{array}{ll}
0 & , \text { if } j<1 \\
1 & , \text { otherwise }
\end{array}, \forall(j)\right.
\end{aligned}
$$

The next two definitions are targeted to LfP's free indices and exempt concepts respectively.

Definition 2.4.2 (LfP's free indices) Let $P$ be a well-formed LfP, then
$P$. free_inds $=U_{(? i d=? a) \in P \text {.args }} a$.free_inds will define the LfP's free indices.
Definition 2.4.3 (Exempt for LfP) Let $P$ be a well-formed LfP, then $P$.exempt(?l) is true iff $\forall((?$ id $=$ ?a) $) \in P$.args $)$ [?a.exempt(? ? ) is true].

### 2.4.1 Local Leaf Problems

The previous definitions demand the LfP not to be local. When local is present in the LfP class declaration all objects instanced from that class will be called local leaf problems $\left(\operatorname{LfP}^{\mathcal{L}}\right)$. There will be a change in $\operatorname{LfP}^{\mathcal{L}}$ argument treatment. Local properties will be defined at a LfP ${ }^{\mathcal{L}}$ level, instead of a global level as has been done until now.

Since we will not have a global declaration for the properties used in a $\operatorname{LfP}^{\mathcal{L}}$, there must be a way of expressing all the necessary information within the $\mathrm{LfP}^{\mathcal{L}}$. For this purpose we will use a different argument declaration,

$$
\text { Listing 2.14: } \operatorname{LfP}^{\mathcal{L}} \text { specification }
$$

```
?id:?class
[
    args }->\mathrm{ [?props, ?free_inds]
]
```

where,

- ? id is a unique atom
- ?class :: d_LeafProblem_C[not abstract, local ] is a non abstract local leaf problem class,
- ?props is a dictionary and ? free_inds is a list of indices.

In the case of $\operatorname{LfP}^{\mathcal{L}_{S}}$ the well-formedness rules change drastically. Since all the data will be defined a priori (in the data file) then the $\mathrm{LfP}^{\mathcal{L}}$ only needs to declare what it wants to export to the rest of the formulation. At least there should be a variable declaration to communicate the results.

Definition 2.4.4 ( $\operatorname{LfP}^{\mathcal{L}}$ well-formedness) $\mathrm{ALfP}^{\mathcal{L}} P_{k}$ is well-formed iff

1. ?id is a unique atom,
2. ?class :: d_LeafProblem_C[not abstract, local ],
3. ?props is composed of elements of the form ? id $=$ ? $p$ (?ind_dims), where ?p is an identifier (unique in ?props) and ?ind_dims is a tuple made of elements of the form ?ind\$?dim or tuples of them, where ? ind and ?dim are atoms (representing an index and a dimension, respectively). In case one of the elements of ?ind_dims is a tuple of ?ind\$?dim, all ?inds and ?dims must be globally declared indices and dimensions, respectively.
4. for each $(? \mathrm{id}=\mathrm{p}($ ? ind_dims $)) \in$ ? props,
(a) if ?class .? args[? id ] $\in\{$ d_var, d_constant, d_expr $\}$ then $\mid$ ?ind_dims $\mid=0$;
(b) if ?class .? args [? id ] = d_vector(?K, ?S) then $\mid$ ? ind_dims $|=|$ ?S $\mid$.
5. ? free_inds is a list of indices such that $\forall(i \in$ ? free_inds ) [d_index( $i$, ?dim) and d_dimension(?dim) (global dimension)].

The list of arguments in a $\operatorname{LfP}^{\mathcal{L}}$ is divided in two sub-lists, ?props and ? free_inds . ?props contains all the scalar and/or vector declarations. Instead of using the mechanism used in LfPs, $\operatorname{LfP}^{\mathcal{L}}{ }_{\mathrm{S}}$ declare local variables and constants. ?props contains all the information regarding property name, indices and dimensions to be created automatically by the system.

On the other hand ? free_inds is a list of indices disjoint to the set specified in ?props; moreover, they must be declared globally.

For example lets take the $\operatorname{LfP}^{\mathcal{L}}$ class d_LP_MPS_C (see Listing 1.7), and build an object from it, Listing 2.15: $\operatorname{LfP}^{\mathcal{L}}$ example

```
ex2 : d_LP_MPS_C
```

[
args $\rightarrow$ [
[
$x=x(i 1 \$ d 1)$,
$c=c(i 1 \$ d 1)$,
$A=A(j 1 \$ d 2, \quad i 1 \$ d 1)$,
$\mathrm{b}=\mathrm{b}(\mathrm{j} 1 \$ \mathrm{~d} 2)$
[],
[]
]
].

The list ?ind_dims can be also specified using tuples, for example, $\mathrm{A}(\mathrm{i} \$ \mathrm{~d} 11, \mathrm{j} \$ \mathrm{~d} 2)$, $\mathrm{k} \$ \mathrm{~d} 3)$. In this case the cardinality of ?ind_dims is 2 . Once the property is extracted from the local file, the size of the dimension linked to the first pair, must be equal to $\mathrm{d} 1 * \mathrm{~d} 2$.

The definitions of free indices and exempt also change.
Definition 2.4.5 ( $\operatorname{LfP}^{\mathcal{L}}$,s free indices) Let $P$ be a well-formed $\operatorname{LfP}^{\mathcal{L}}$, such that $P$.args[?_,? fi], then $P$.free_inds $=$ fi, defines the $\operatorname{LfP}^{\mathcal{L}}$ 's free indices.

Definition 2.4.6 (Exempt for $\operatorname{LfP}^{\mathcal{L}}$ ) Let $P$ be a well-formed $\operatorname{LfP}^{\mathcal{L}}$, such that $P$.args[?_? fi] then $P$.exempt(?l) is true iff ? $\subseteq \subseteq$ ?fi .

## LfP $^{\mathcal{L}}$ 's semantics

Once the system encounters a $\operatorname{LfP}^{\mathcal{L}}$ declaration, besides verifying its well-formedness it makes some preprocessing.

Let free_inds be the list of free indices of a certain $\operatorname{LfP}^{\mathcal{L}}$ and free_dims the corresponding list of dimensions, then for each term $\mathrm{p}\left(\operatorname{ind}_{1} \$ \operatorname{dim}_{1}, \ldots\right.$, ind $\left._{m} \$ \operatorname{dim}_{m}\right)$ found in the list props of $\operatorname{LfP}^{\mathcal{L}}$ the system creates the following property globally, (assume the $\mathrm{LfP}^{\mathcal{L}}$ 's identifier is pll )

Listing 2.16: Automatic property generation

```
p(p|l) : VC.
```

p(pll) : d_property
[
dims $\rightarrow$ free_dims $\cup\left[\operatorname{dim}_{1}, \ldots, \operatorname{dim}_{m}\right]$
].
and for each $\operatorname{dim}_{i}$ and $\operatorname{ind}_{i}(i \in[1 . . m]$, the system will create the following dimension and index
Listing 2.17: Automatic dimension and index generation

```
d_dimension( dim}\mp@subsup{i}{i}{\prime},\textrm{pll})
d_index(indi, 年mim).
```

where VC is either d_var or d_constant depending on the corresponding parameter type in pll's class.
For example, for the $\operatorname{LfP}^{\mathcal{L}}$ ex2 declared in the previous subsection, this is part of predicates that the system will automatically generate,

Listing 2.18: Example of property and dimension generation

```
d_dimension(d1, ex2).
d_index(i1, d1).
x(ex2) : d_var.
x(ex2) : d_property
```

```
[
    dims -> [d1]
].
```

Properties defined in a $\operatorname{LfP}^{\mathcal{L}}$ can be accessed from other components of the model making reference to its full name, for instance, $\times(\mathrm{ex} 2)(\mathrm{i1})$. Note that the name of the property changes to be accessed globally by including the name of the component that declares it.

### 2.5 Blocks

Blocks will be used to represent composition of structures. A block in I-DARE(im) is defined as follows,

Listing 2.19: Block definition

```
?id:?class
[
    subs -> ?subs,
    subVP -> ?subVP,
    [freel -> ?freel]
].
```

where,

- ?id is a unique atom,
- ?class ::d_Block_C[not abstract] is a non abstract block class,
- ?subs is a list composed of atoms or tuples of atoms,
- ?subVP is a list of variable patterns (see $\S 1.2 .3$ ),
- ? freel is a list of indices. optional.

Assume we have a method vars that can be applied to any component and returns the set of all variable identifiers used within the component. Let us also assume that when freel is not present in the block's declaration then ? freel $=\emptyset$ and that each substructure is considered a tuple (note that only one structure is the tuple with one element).

Definition 2.5.1 (Block well-formedness) A block is well-formed iff

1. ?id is a unique atom,
2. ?class :: d_Block_C[not abstract],
3. $\mid$ ?subs $|=|$ ?class. subsC $|=|$ ?subsVP $\mid$,
4. ?subs $[\mathrm{i}]_{l}$ ??class .subs $\mathrm{C}[\mathrm{i}][$ wellformed $]$,
5. ?subs $[\mathrm{i}]_{l}[$ exempt(?freel ) $]$,
6. ?subs $[\mathrm{i}]_{l}$.vars $\cap$ ?subs $[\mathrm{i}]_{l}$-vars $=\emptyset$,
7. $\forall \mathrm{i} \neq \mathrm{j} \in[0 . \mid$ ? $\mathrm{subs} \mid]$, ?subs $[\mathrm{i}] \neq$ ?subs $[\mathrm{j}]$,
8. ?subVP $[\mathrm{i}] \subseteq \cup_{l}$ ? subs $[\mathrm{i}]_{l}$.vars,
9. ?subVP must unify with ?class. link ensuring the DNU rule (see definitions 1.2.9 and 1.2.10),

When constructing a block the previous well-formedness rules must be guaranteed (and will eventually be verified). Rules from 2 to 5 are related with the cardinalities, well-formedness and exempt of the block's sub-components. Each sub-component must be well-formed, different pair to pair and exempt in the list? freel.

The second group of rules are the ones related to the link (i.e. how sub-components' variables will interact). Note that with this rule we enforce that we can not include new variables (i.e. variables not used in the sub-components) and we also define the way these variables will communicate between each other.

A block uses its freel list to control the inner replication of its sub-components, for example if we have a block $A$ with no freel, such that ex2 $\in A$.subs, assume we add a dimension d3 as free index of ex2; then in the final instance ex2 will be replicated $\|\mathrm{d} 3\|$ times (see figure 2.2).


Figure 2.2: Replication inside blocks
On the other hand, when a block $B$ defines a list of free indices (in freel), it means that $B$ may be replicated inside a parent block. Let's assume $B$ will be replicated $k$ times, then $B$ needs all its substructures variables to be partitioned in $k$ disjoint subsets (i.e. there is no variable that links one replication of $B$ with the other). This fact is ensured with the condition 5 of the block's well-formedness definition.

The following examples will illustrate how blocks can be defined. Assume we have the following d_LP_C and d_IC_C instances.

Listing 2.20: LP and Integrality constraint

```
ex3 : d_LP_C
[
    args -> [
            x = $([$(x(k1),[k1]), $(y(k2),[k2])])
            c=$(1, [i]),
            A = $(a(j,i), [i,j]),
            b = $(-cs([0->(j < 1), 1]), [j]),
            rels = $('=<', [j]),
            dir = 'min'
            ]
].
ic : d_IC_C
[
    args -> [$(y(k2), [k2])]
].
```

In ex3 definition we use a K-vector (line 4), because variables will be divided in integer and real. Note that when creating the instance, I-DARE will verify that $\mathrm{k} 1+\mathrm{k} 2=\mathrm{i}$.

Using ex3 and ip we can create an instance of block class d_B_MILP_C.
Listing 2.21: MILP block instance

```
milpB : d_B_MILP_C
[
    subs -> [ex3, ic ],
    subVP -> [[y,x], [y]]
].
```

We can build another example using d_B_vardept_C block class

Listing 2.22: Another block example

```
block1 : d_B_vardept_C
[
    subs -> [ex3, ic ],
    subVP }->[[y,x], [y,()]
].
```

Note the usage of () to represent the second group of variables of ic. Since d_B_vardept_C defines that for its second argument the template must be $[\mathrm{X}, \mathrm{W}]$, the exported variables must have the same cardinality. In this case the usage of the empty variable tuple () is allowed because W must be disjoint to V (from the first template $[\mathrm{X}, \mathrm{V}]$ ).

The two previous examples use a particularity of the global structures, that is being able to use K-Vectors. Note that the parameter $\times$ of ex3 is built using a K-Vector that concatenates the global variables $x$ and $y$. This variable partition allows easily to create the milpB block's variable patterns. Assume we wanted to create a MPS representation of the MILP block. One possible solution is to use d_LP_MPS_C as a substructure. However, since MPS has a way of distinguishing integer from real variables ('MARKER' 'INTORG' and 'MARKER' 'INTEND'), we could create a $\operatorname{LfP}^{\mathcal{L}}$ class for MILP that uses directly the MPS format.

Listing 2.23: MILP MPS $\operatorname{LfP}^{\mathcal{L}}$ class

```
d_MILP_MPS_C :: d_LeafProblem_C
[
    dim_var >> [cons, colsR, colsl],
    args -> [
        xr = d_vector(d_var, [colsR]),
            xi = d_vector(d_var, [colsl]),
            cr = d_vector(d_constant, [colsR]),
            ci = d_vector(d_constant, [colsl]),
            Ar = d_vector(d_constant, [cons, colsR]),
            Ai = d_vector(d_constant, [cons, colsl]),
            b = d_vector(d_constant, [cons]),
            rels = d_vector(d_rel, [cons]),
            dir = d_direction
            ],
    local
].
```

This $\operatorname{LfP}^{\mathcal{L}}$ class represents a MILP with the data in MPS format, note that the cols dMV (of d_LP_C) is split into two dMVs, one for the columns related to real variables and the other for the columns related with integer values. Of course, we could also create a LfP class for MILP (d_MILP_C) very similar to d_MILP_MPS_C. Note that for each LfP we might create respective LfP ${ }^{\mathcal{L}}$ s representing specific formats. Therefore, a big amount of structure classes may be created, making difficult the selection of structures while building the model. In chapter 5 we will see how we can deal with this problem from a reformulation viewpoint, sometimes abstracting the user from the particular formats.

The following definitions will specify how free index and exempt are done in blocks.

Definition 2.5.2 (Block's free indices) Let $B$ be a well-formed block, then $B$. freel defines the block's free indices.

When generating the instance if freel is not present, the block will replicate all the inner components covering all their free indices.

Definition 2.5.3 Let $B$ be a well-formed block, then $B$ [exempt(? $)$ ] is true iff $B$. freel $=$ ? ।.
See that exempt verification for blocks is pretty simple, since a block ensures exemption of all its substructure's variables based on its freel list.

### 2.6 Formulation

Using the previously defined constructs we can build a formulation of a problem. From now on we will assume that when talking about components we will be talking about well-formed components.

A formulation will be an instance of the following class,
Listing 2.24: Formulation class

```
d_Formulation
[
    dimensions => _list ,
    indices => -list,
    properties => _list ,
    root }\quad=>\mathrm{ d_Component_C
].
```

where

- dimensions - is a list of well-formed dimensions,
- indices - is a list of well-formed indices,
- properties - is a list of well-formed properties,
- root - is an instance of d_Component_C such that root[wellformed] is true.

A formulation does not explicitly store all the components, instead it only needs to know the root component and due to the well-formedness rules all the other components must be contained in a tree structure starting from root.

When creating a formulation we must ensure that there are no repeated components. Even if we ensure that all sub-components of a block are different, it is not enough to ensure that all components starting from root are different. For example, assume the component composition exposed in figure 2.3.


Figure 2.3: An example of wrong component tree
Observe that C, B and A satisfy the rule of not repeated sub-components. However, we can see that $E$ is repeated in $C$ and $B$. This is because, a block check just for its immediate sub-components. Then, at a formulation level there can appear component repetitions. Therefore we need enforce a global component non-repetition in the formulation well-formedness rule.

Definition 2.6.1 (Formulation well-formedness) A formulation $\Phi$ is well-formed iff

1. $\forall d \in \Phi$.dimensions, $d$ is well-formed,
2. $\forall p \in \Phi$. properties, $p$ is well-formed,
3. $\forall i \in \Phi$. indices, $i$ is well-formed,
4. $\Phi . \operatorname{root}[$ wellformed] is true, and
5. if $\Delta$ is the set of all $\delta$ : d_Component_C $\in \Phi$ (deductible from root) then $\forall\left(\delta, \delta^{\prime} \in \Delta\right)\left[\delta \neq \delta^{\prime}\right]$ (no cycle in the components' tree).

Putting together some of the examples we presented in previous sections, we may build a complete formulation,

Listing 2.25: Full formulation example

```
//** Dimensions
d_dimension(d1).
d_dimension(d2).
d_dimension(d11).
d_dimension(d12).
//** indices
d_index(i, d1).
d_index(j, d2).
d_index(k1, d11)
d_index(k2, d12).
//** Properties
a : d_constant.
a : d_property
[
    dims -> [d2, d1]
].
x : d_var.
x : d_property
[
    dims -> [d11]
].
y : d_var
y : d_property
    dims -> [d12]
    upper }->\mathrm{ - 1,
    lower -> 0
].
//** Leaf Problems
ex3 : d_LP_C
[
    args -> [
        x = $([$(x(k1),[k1]), $(y(k2),[k2])])
            c}=$(1, [i])
            A = $(a(j,i), [i,j]),
            b = $( _cs([0-> (j<1), 1]), [j]),
            rels = $('=<', [j]),
            dir = 'min
            ]
].
ic : d_IC_C
[
args -> [$(y(k2), [k2])]
].
//** Blocks
milpB : d_B_MILP_C
[
    subs -> [ex3, ic ],
    subVP -> [[y,x], [y]]
].
```

```
//** Formulation
form1 : d_Formulation
[
    dimensions -> [d1, d2, d11, d12],
    indices -> [i, j, k1, k2],
    properties }->\mathrm{ [x, y, a],
    root }\quad>\mathrm{ milpB
] .
```

Note that in this example the variable $y$ it is not just integer, but also binary, due to the specified bounds.

### 2.7 Discussion

The internal model (IM) is based on a set of defined structure classes (I-DARE(lib)), it offers an expressive way of constructing structured well-formed models. I-DARE(im) defines all the constructs that allow us to verify for well-formedness of an incoming formulation. Moreover, I-DARE(im) offers an initial set of tools for querying the IM, we could ask for free indices of variables, leaf-problems, blocks; or we could retrieve whether a component is going to be replicated or not. Queries may become more complex depending on the developers needs.
$\mathcal{F}$ LORA-2 plays an important part in I-DARE(im), because it allows us to define all the syntactic constructs together with the semantic using a formal language that at the same time enables us to implement all those definitions.

A declarative definition of the problem's formulation, enables us to construct a set of wrappers around the formulation to allow data manipulation, increasing the amount of information that can be queried (this is done in package I-DARE(ei)). This means that the model will become a huge source of organized information.

Building and solving a model is not I-DARE's only goal, as it is for many modeling environments. With I-DARE we are interested in creating an environment that allows us to build a model, to exploit all the information that model can bear, and use that information for reformulating it and finally solving it. The usage of declarative programing in the design of I-DARE(lib) and I-DARE(im) permits us to take advantage of that information and also leaves the door opened for other developers that may come up with more powerful queries.

With the formal definition of well-formed Formulation, we lay the foundations for creating the reformulation package. While defining this package we will see how querying the model+data becomes crucial.

## Chapter 3

## I-DARE(ei) - the enhanced instance


#### Abstract

$\qquad$ Once the internal model is built and verified for well-formedness, we need to construct the instance to feed it to the solvers. I-DARE introduces a type of instance called enhanced instance (I-DARE(ei)) which uses the structural information contained in the internal model.

This chapter will define I-DARE(ei) as a wrapper to the enhanced model that will provide information regarding to the data and also will generate the enhance instance data file structure that will be used by the solving part of I-DARE.


### 3.1 Instance Wrapper (IW)

Once we build the internal model and verify it for well-formedness, we need to construct an instance of this model using the problem data. For this purpose, I-DARE provides a module called I-DARE(ei). I-DARE(ei) defines a formulation's wrapper class called d_InstanceWrapper, that defines the guidelines of how the data must be arranged to eventually generate the final instance.

The d_InstanceWrapper class allows the system to access the data related with dimensions and scalars of global and local components. This is the class definition,

Listing 3.1: Instance Wrapper
[

```
```

d_InstanceWrapper

```
```

d_InstanceWrapper
// Properties
// Properties
formulation }\quad=> d_Formulation
formulation }\quad=> d_Formulation
global }\quad=>\mathrm{ d_DataHandler,
global }\quad=>\mathrm{ d_DataHandler,
local(?prob) }\quad=>\mathrm{ _list, // of Data Handlers
local(?prob) }\quad=>\mathrm{ _list, // of Data Handlers
// Methods
// Methods
dimSize(?dn) => _integer, // Gets a dimension's size
dimSize(?dn) => _integer, // Gets a dimension's size
constVal(?icons) => _double, // Gets a constant value
constVal(?icons) => _double, // Gets a constant value
upperBnd(?ivar) => _double, // Gets the upper bound of a variable
upperBnd(?ivar) => _double, // Gets the upper bound of a variable
lowerBnd(?ivar) => _double // Gets the lower bound of a variable
lowerBnd(?ivar) => _double // Gets the lower bound of a variable
].

```
```

].

```
```

formulation - must be a property that sets which formulation the wrapper will handle.
global - must be the Data Handler that will deal with the global data (see §3.1.2).
local (?prob) - must be a list of Data Handlers that will deal with the data of the LfP ${ }^{\mathcal{L}}$ ?prob:d_LeafProblem_C[local].
dimSize(?dn) - retrieves the cardinality of a certain dimension ?dn (see $\S 3.1 .4$ for a more accurate definition).
constVal(?icons) - retrieves the value of a certain indexed constant ?icons (see §3.1.4).
upperBnd(?ivar) - retrieves the upper bound of a certain indexed variable ? ivar (see §3.1.4).
lowerBnd(?ivar) - retrieves the lower bound of a certain indexed variable ?ivar (see §3.1.4).
In the definition of the d_InstanceWrapper we used d_DataHandler. A d_DataHandler is an interface that abstracts the format in which the data is represented. Also, note that we used a property global and a property local. Global refers to all the data related with LfP, whereas local refers to the data handlers linked to the $\mathrm{LfP}^{\mathcal{L}}$.

The parameterized property local(?prob) must define a list composed of objects DH such that DH:d_DataHandler and DH is registered to the class of ?prob (see §3.1.2).

To specify the properties local (?prob) must guarantee, we must first make some definitions,
Definition 3.1.1 (Single Dimension-Index Cardinality) Let ? IW be an instance wrapper and $d i$ be a dimension or an index, then di's cardinality will be defined by

$$
\|d i\|= \begin{cases}\text { ?IW.dimSize }(d i) & \text { if d_dimension }(d i) \text { or d_dimension }(d i, ? \mathrm{plI}) \\ \text { ?IW.dimSize(?d) } & \text { if d_index }(i d, ? \mathrm{~d}) \text { and (d_dimension(?d) or d_dimension(?d,?plI)) }\end{cases}
$$

Definition 3.1.2 (Multiple Dimension-Index Cardinality) Let $D I$ be a list of dimensions and/or indices, then $D I$ 's cardinality will be defined by,

$$
\|D I\|=\prod_{d \in D I}(\|d\|)
$$

Having the previous definitions we can specify the property local(?prob) must always ensure that given an IW:d_InstanceWrapper,

$$
\mid \text { IW. local(?prob) } \left\lvert\,= \begin{cases}\| \text { ?prob.freeinds } \| \text { or } 1 & \text { if } \mid \text { ?prob.freeinds } \mid>0 \\ 1 & \text { otherwise }\end{cases}\right.
$$

This rule forces the user to supply a data handler for each replication the $\mathrm{LfP}^{\mathcal{L}}$ or one data handler common to all replications of the same $\operatorname{LfP}^{\mathcal{L}}$.

### 3.1.1 Global Data

Building the model of a problem is not enough, in order to solve the problem we also need the data that will instantiate the model pieces, say dimensions, constants, leaf problems and blocks.

Since I-DARE(im) give the possibility to explicitly define LfP and $\operatorname{LfP}{ }^{\mathcal{L}}$ s, the data will be divided in two groups, global data and local data.

Data for dimensions and properties explicitly declared in the internal model (i.e. not the ones generated by the system) will be provided in a global data file. This file must contain the sets composed by the dimensions' cardinalities and the values of the constants.

I-DARE(ei) provides us with a default format for defining a global data file based on XML. This format will be defined as follows,

Listing 3.2: XML global data format

```
<DIMS>
    <DIMENSION name="dim_name" card=" cardinality_of_dimension"/>
</DIMS>
<CONS>
    <CONSTANT name=" constant_name" default="optional_default_value">
        <VALUE dim="dim_val_1,..., dim_val_n" value="constant_value" />
    </CONSTANT>
</CONS>
```

where

- dim_name - is the global dimension's name in the internal model,
- cardinality_of_dimension - specifies the cardinality of the dimension,
- constant_name - is a name of a constant property in the internal model,
- dim_val_i - is an integer number that has to correspond to the proper dimension specified for the property in the internal model,
- constant_value - is a double that has to satisfy the bounding of the corresponding property,
- optional_default_value - is a double that will be assigned automatically by the system to the indexings not present in the file. If the optional_default_value is not present the system will assume 0 as default value.

For example, say we have the following piece of internal model,

```
dimension(d1).
dimension(d2).
b:d_constant.
b:d_property
[
    dims -> [d1]
].
a:d_constant.
a:d_property
[
    dims -> [d1, d2]
].
```

One possible XML global data file for that model could be,

```
<DIMS>
    <DIMENSION name="d1" card=" 3" />
    <DIMENSION name=" d2" card=" 2" />
</DIMS>
<CONS>
    <CONSTANT name="b">
        <VALUE dim="0" value=" 2.5"/>
        <VALUE dim="1" value="1" />
        <VALUE dim="2" value=" 5.67" />
    </CONSTANT>
    <CONSTANT name="a">
        <VALUE dim=" 0,0" value=" 0.2" />
        <VALUE dim=" 1,0" value=" 0.6"/>
        <VALUE dim=" 1,1" value="-2" />
        <VALUE dim=" 2,1" value="3"/>
    </CONSTANT>
</CONS>
```

Note that for the instanced indexed constants $a(2,0)$ and $a(0,1)$ the value will be 0 (default by omission).

On the other hand, when we are in presence of a $\operatorname{LfP}{ }^{\mathcal{L}}$, we will need to handle local data. Local data contains information that may appear in a wide variety of formats depending on the structure we are handling. For example, the LfP ${ }^{\mathcal{L}}$ class d_LP_MPS_C, defines a class of LP which data comes from a MPS file [97].

There is one thing in common between global and local data, it is that for both of them we will need a data handler. Note that in the end, the XML file for global data could be seen as a specific format. It is also true that it is a format highly influenced by the information required by
the global part of the internal model. Therefore the XML data handler will generally be used to deal with global data. The following subsections will define how a Data Handler can be designed and implemented.

### 3.1.2 Data handler (DH)

There must be a unique interface in I-DARE(ei) to read data files, regardless of the format in which the data is written. This interface is d_DataHandler and is defined as follows,

Listing 3.3: DH Class

```
d_DataHandler
[
    // Methods
    => general_check(?S),
    => fd_check(?fdS)
    dimSize(?dn) => _integer,
    constVal(?icons) => _double,
    upperBnd(?ivar) => _double,
    lowerBnd(?ivar) }\quad=>\mathrm{ _double,
    dataTag(?IW, ?C, ?ID) => _list
].
```

Note that more than the half of the methods have the same signature to the ones present in d_InstanceWrapper. This means that d_InstanceWrapper makes delegation on the d_DataHandlers when accessing the data.

The DH class introduces three new methods,
general_check (?S) - must be implemented to make initial verifications and initializations depending on the structure ?s.
fd_check(?fdS) - must control whether the DH can provide different data instances for ?fdS replications of the $\mathrm{LfP}^{\mathcal{L}}$. Note that if the structure is global, no verification must be done.
dataTag(?IW, ?C, ?ID) - must generate a two elements list, the first being a list of XMLNodes (following the XSB Prolog specification [147]), and the second is the list of instanced indexed variables inside the component ?C. ?ID is a dictionary composed of pairs index = val for all indices external to the $\operatorname{LfP}^{\mathcal{L}}$ (?c), that enables the DH to instantiate the variables. The XMLNode returned by this method contains the information of how the data will be represented (see MPS example in following sections).

## Registration and some examples

Data handlers must be subscribed to the structures they can be applied to. The subscription procedure will take place using the following predicate,

Listing 3.4: Data Handler subscription predicate
d_DHsubscription (?structure_c, ? DH_c, ?dim_dict, ?prop_dict).
where,

- ? structure_c - is the atom global or a $\operatorname{LfP}^{\mathcal{L}}$ class name (i.e. ? structure :: d_LeafProblem_C[not abstract, local ]),
- ?DH_c - is a DH class (i.e. ?DH_c::d_DataHandler),
- ?dim_dict and ?prop_dict - are dictionaries that depend on the DH and the structure classes. This dictionaries must resolve name differences between the actual data file and the structure's class we are planning to handle (see examples bellow).

For example, we may create a DH class to deal with the XML data file presented in §3.1.1.

Listing 3.5: XML Data Handler Class

```
d_XMLDataHandler :: d_DataHandler
[
    //Properties
    xmlfile => _string
].
d_DHsubscription(global, d_XMLDataHandler, [], []).
?X:d_XMLDataHandler[general_check(?S)] :-
    ?X[initcheck(?S)],
    ?X[xmlfile -> ?filename],
    load_XML(?filename, ?data),
    insert{?X[data }->\mathrm{ ?data]}.
///.... Rest of the methods
```

In the general_check (?S) method we first call initcheck (?S) that will verify whether ? $\mathrm{S}=$ global and will look for the subscription. Later on, other verification are done, by checking for the XML file existence and by loading the data present in that file.

Note the for the particular case of d_XMLDataHandler the method fd_check(?fdS) is not defined, since the data the DH manipulates will always be global data. Moreover, the dictionaries in the subscription are both empty ([ ]). This can be done because the names in the XML file correspond one to one to the names in the model (see $\S 3.1 .1$ ).

Lets see how an IW can be created for the formulation form1 in Listing 2.25 in the previous chapter,

Listing 3.6: IW example

```
gxml:d_XMLDataHandler
[
    xmlfile -> 'data.xml'
].
iw1:d_InstanceWrapper
[
    formulation -> form1,
    global -> gxml
].
```

Since form1 has no $\operatorname{LfP}^{\mathcal{L}}$ in its structures' tree, there is no need for iw1 to use the local (...)->... properties to assign the DHs of the $\mathrm{LfP}^{\mathcal{L}}{ }_{\mathrm{s}}$.

On the other hand, we may define a handler to deal with MPS data files, which can be subscribed to structure classes like d_LP_MPS_C or d_MILP_MPS_C,

Listing 3.7: MPS Data Handler Class

```
d_MPSDataHandler::d_DataHandler
[
    MPSfile => _string
    //Class Properties
    dims -> ['rCOLS', 'CONS', 'iCOLS'],
    props -> ['rOF', 'rMATRIX', 'rVARS', 'RHS', 'RELS', 'DIR', 'iOF', 'iMATRIX', 'iVARS']
].
d_DHsubscription(d_LP_MPS_C, d_MPSDataHandler,
    [cols='rCOLS', cons='CONS'],
    [c='rOF', x='rVARS', A='rMATRIX', b='RHS', rels='RELS', dir='DIR']).
d_DHsubscription(d_MILP_MPS_C, d_MPSDataHandler,
    [colsR='rCOLS', colsl='iCOLS', cons='CONS'],
    [cr='rOF', ci='iOF', xr='rVARS', xi='iVARS',
    Ar='rMATRIX', Ai='rMATRIX',
    b='RHS', rels='RELS', dir='DIR']).
```

19
20
//.. verification and data retrieval methods

Note that in this case, the d_MPSDataHandler defines the set of dimension names and property names that are independent from the ones in the structure classes it may be registered to. In fact, in the subscriptions can be seen how these names are matched to the ones in the structure classes.

Since the MPS data file may represent a pure LP or also a MILP, the d_MPSDataHandler divides the columns into real and integer (using the prefixes $r$ and $i$, respectively). In the d_LP_MPS_C subscription, it only specifies the name conversions for the real part, but in the d_MILP_MPS_C subscription it uses all of them (real and integer part).

We will not show all verification methods because the code is a little bit long. However, we would like to stop a second at the fd_check(?fdS) method, because d_MPSDataHandler is used for local data (as every other DH except for $d_{-} X M L D a t a H a n d l e r$ ). For a DH assigned to a certain $\operatorname{LfP}^{\mathcal{L}}$ (i.e. local(?PII) -> ?DHL) there are three possible scenarios,

1. $\|$ ?PII. freeinds $\|=1$ or ? PII . freeinds $=\emptyset$ - then there is no replication of ?PII, therefore ? $\mathrm{DHL}=[? \mathrm{DH}]$, where ?DH must be a data handler subscribed to ?PII.
2. \|?PIII. freeinds $\|>1$ - then
(a) if $\mid$ ?DHL $=\|$ ?PII.freeinds $\|$ - then all ? $\mathrm{DH} \in$ ?DHL will be used to read one instance of ?PII.
(b) if |?DHL|=1 - then the system will ask the only ?DH $\in$ ?DHL to fd_check(||?PII. freeinds ||),
i. The ?DH checks for and supplies \|?PII. freeinds \| different instances
ii. or the ?DH has only one instance that will be shared to all replications of ?PII.

Notice that when only one DH is specified in the local section of a certain $\mathrm{LfP}^{\mathcal{L}}$, the fd_check(?fdS) is invoked, in order to verify whether the DH internally provides the right number of different data instances. For example, a MPS file may contain more than one RHS vector. This fact may be used to store several different data instances in a single MPS file (changing the RHS parts of the equations). So, the d_MPSDataHandler must verify in its fd_check(?fdS) method whether the number of RHS vectors is 1 or ?fdS.

## Data Access

One of the two main functionalities of a DH is to access the data. For doing so they define certain methods that allow the retrieval of dimension cardinalities, constant values and variable bounds. This data access is done through four methods,

- dimSize(?dim) -> ?size,
- constVal(?icons) -> ?val,
- upperBnd(?ivar) $->$ ?bnd and
- lowerBnd(?ivar) -> ?bnd.

For the global DH these methods will be called using the names defined in the model. For instance, if there is a dimension, declared d_dimension(plant), then to know the size of plant, one will call ?gDH[dimSize(plant) -> ?size], where ?gDH is the global DH. For constants values is essentially the same, just that the indexed constant we pass as argument, must have the indices instantiated (e.g. instead of $A(j, i)$, we must substitute $i$ and $j$ by values in $[0 . .\|i\|-1]$ and $[0 . .\|j\|-1]$, respectively).

The bound methods will be completely ignored in a global DH , since the bounds will be deduced from the model (and the constant values).

On the other hand, for the rest of DH handlers (the ones used for $\operatorname{LfP}^{\mathcal{L}} \mathrm{s}$ ) the arguments' meaning will slightly change. This change will be in part influenced by the fact that there may be a different DH for each replication of the $\mathrm{LfP}^{\mathcal{L}}$. Since the amount of replications of a $\operatorname{LfP}^{\mathcal{L}}$ is $\| L f P^{\mathcal{L}}$.freeinds $\|$, the free indices of that $\operatorname{LfP}{ }^{\mathcal{L}}$ will play a crucial part while determining which DH will be used.

Definition 3.1.3 (Instanced indices) Let $I$ be a list of indices then the set of instanced indices is defined by,

$$
I^{\xi}=\left\{\left(I_{1}^{v}, \ldots, I_{|I|}^{v}\right) \mid I_{j}^{v} \in\left[0 . .\left\|I_{j}\right\|-1\right], \text { with } j \in[1 . .|I|]\right\} .
$$

The set of instanced indices defines the set of all possible combinations of values assigned to indices. Notice that for a list of indices $I,\left|I^{\xi}\right|=\|I\|$.

Definition 3.1.4 (Linearization function) Let $I$ be a list of well-formed indices, and $\left(I_{1}^{v}, \ldots, I_{|I|}^{v}\right) \in$ $I^{\xi}$ then

$$
\zeta\left(I_{1}^{v}, \ldots, I_{|I|}^{v}\right)=\sum_{j \in[1 . .|I|]}\left(I_{j}^{v} * \prod_{k \in[j+1 . .|I|]}\left(\left\|I_{k}\right\|\right)\right)
$$

will defined the linearization function of a list of indices.

For example assume we have indices i and j with sizes 3 and 5 respectively. If we instance the indices $\mathrm{i}=1$ and $\mathrm{j}=2$ then $\zeta(1,2)=5$, being 5 the linear position. Note that, to linearize a two dimensional position is to see the matrix as big sequence of concatenations of that matrix's rows. The $\zeta$ definition generalize that concept to $n$ dimensions.

Assume we have a $\operatorname{LfP}^{\mathcal{L}}$ of class ?C, with free indices ? fil, then for the access methods,

- dimSize(?dim) -> ?size,
- constVal(?icons) -> ?val,
- upperBnd(?ivar) $->$ ?bnd and
- lowerBnd(?ivar) $->$ ?bnd.
we have that,
- ?dim - must be a term of the form ?dname $\left(\zeta\left(f i_{1}, \ldots, f_{n}\right)\right)$ where ?dname $\in$ ?C.dim_var and $\left(f i_{1}, \ldots, f_{n}\right) \in$ ?fiL ${ }^{\xi}$. If $n=0$ ?dim must be equal to ?dname.
- ?icons - must be a term of the form ?cname $\left(\zeta\left(f i_{1}, \ldots, f_{n}\right)\right.$, ? $\left.\mathrm{ii}_{1}, \ldots, ? \mathrm{i} \mathrm{i}_{k}\right)$, where ?cname is a parameter identifier of ?C such that (?cname $=$ ? type) $\in$ ?C.args; $\left(f i_{1}, \ldots, f_{n}\right) \in$ ?fiL ${ }^{\xi}$; and
- if ? type = d_vector(?T, ?S) and ?T $\neq$ d_var then $k=$ ? S and (? $\mathrm{ii}_{1}, \ldots$, ? $\mathrm{ii}_{k}$ ) $\in$ ? $\mathrm{S}^{\xi}$,
- if ?type $\neq$ d_vector(?-, ?-) and ?type $\neq$ d_var then $k=0$.

Note that ? cname must correspond to a constant, relation or direction type (or the respective vectors).

- ?ivar - will have the same structure as ?icons, but it must refer to a variable type (or a vector of variables).

The DHs will implement these methods according to its data format and the dictionaries specified in the subscription. For example, d_MPSDataHandler will search for the corresponding dimension, constant or variable name in the subscription dictionaries, and then using the internally loaded data it will provide with the adequate response. Note that the DH may use the ? fi to look for the corresponding internal replication, in the particular case of d_MPSDataHandler, it will look for the corresponding RHS.

### 3.1.3 IW - Data Validation

The instance wrapper will initially ensure that certain properties of the model are still satisfied once the data is present. These properties are mainly the consistency of dimension sizes, vector sizes and constant values.

In the case of global data there are some rules that the XML must satisfy, rules that depend exclusively on the model the XML is linked to.

Definition 3.1.5 (XML - Rules) Given ?F:d_Formulation a well-formed formulation, and a XML global data file (see format in Listing 3.2),

1. $\forall$ (dim_name)[ dim_name $\in$ ? F.dimensions $\wedge$ d_dimension(dim_name)];
2. $\forall$ (constant_name)[ constant_name:d_constant $\wedge$ constant_name:d_property $\wedge$ constant_name $\in$ ?F.properties $\wedge$ optional_default_value $\in$ [constant_name.lower..constant_name.upper] $\wedge \forall\left(\right.$ dim_val $\left._{i}\right)$, with $i \in[1 . . n]$,
(a) $n=\mid$ constant_name.dims $\mid$,
(b) dim_val ${ }_{i} \in[0 . . \|$ constant_name.dims $[i] \|-1]$, and
(c) constant_value ${ }_{i} \in$ [constant_name.lower..constant_name.upper].
3. $\forall$ (constant_name) if there is a ?PI of class ?C::d_LeafProblem_C[not local], for which (?pn = constant_name) $\in$ ?PI.args and (?pn, ?dn) $\in$ ?C.dim_bounds, then $\forall$ constant_value of constant_name (including the default one), constant_value $\in[0 . .\|? \mathrm{dn}\|-1]$.

Note that rule 3 has only effect when ?C.dim_bounds is defined, otherwise it will be ignored.
There is an important rule related to the vector sizes.
Definition 3.1.6 (Vector sizes - Rule) For each ?PI:?C in ?F:d_Formulation, such that ?C::d_LeafProblem_C; let $C S$ be the list composed of the cardinalities of all dMVs in ?S, such that (? id = d_vector(?K,?S)) $\in$ ?C.args; and $v$ be a the vector, such that (? id $=v) \in$ ? PI.args, then the $|v|=C S$.

The size of a vector $|v|$ is computed depending on how it is built. When it is a simple vector the size will be computed based directly on the index list sizes. On the other hand, when it is K-vector the size will be the sum in each dimension of the appended simple vectors.

For instance let's revisit the example in Listing 2.25. In this example we have a LfP structure (ex3) of type d_LP_C,

```
ex3 : d_LP_C
    args }->\mathrm{ [
        x = $([$(x(k1),[k1]), $(y(k2),[k2])])
        c=$(1, [i]),
    A=$(a(j,i),[i,j])
    b}=$(-cs([0->(j<1),1]), [j])
    rels=$('<', [j]).
    dir = 'min'
].
```

```
d_LP_C :: d_LeafProblem_C
    dim_var }->\mathrm{ [cols, cons],
    dim_var 
        [ x = d-vector(d_var, [cols]),
        A = d_vector(d_constant, [cons, cols])
        b = d_vector(d_constant, [cons]).
        rels=d_vector(d_rel, [cons]),
            dir = d_direction
        ] d
].
```

Once the formulation is linked to the data, cols and cons will have an actual size, that in this case will depend on $|\mathbf{k} 1|+|\mathbf{k} 2|$ and $|i|$ (for cols), and on $|j|$ (for cols). The vector sizes' rule will ensure this relation in all the vectors in ex3. Using classic unification mechanism, the system will deduce the sizes of cols and cons and verify that all the sizes are correct depending on the vectors' disposition within d_LP_C.

The Vector sizes' rule (3.1.6) is of particular interest in the case of LfPs, because they have a more rich mechanism of constructing vectors that may provoke an inconsistency of the rule. For $\mathrm{LfP}^{\mathcal{L}}$ s, ensuring this rule is more simple, since the way vector parameters are specified is very close to the type representation. However, when we use global dimensions in the $\operatorname{LfP}^{\mathcal{L}}$,s vector specification there must be a correspondence between the expected size and the global dimension's cardinality (or the product if more than one).

This last assessment take us to this rule.

Definition 3.1.7 (Mixed dimensions - Rule) Let ?d be a dimension such that d_dimension(?d) and d_dimension(?d, ?p) then the global ?d size must be equal to the size of ?d extracted from the DH assigned to ?p.

Since, a dimension may be shared between the local and global parts of the data, the system must verify a shared dimension's consistency (no multiple cardinalities). This validation is done applying the Mixed dimensions' rule.

Involving dimensions and blocks, there is another thing we have to verify. When a block class defines the rpIR (replication relations) field, we need to ensure that the amount of replications of the substructures inside the block corresponds to the expressions specified in rpIR. For doing so we use the following rule,

Definition 3.1.8 (Replication Relations - Rule) Let ?b be a block, such that ?b:?Cb, ?Cb::d_Block_C, then for all $(? \mathrm{id}=?$ expr $) \in ?$ Cb.rpIR,

$$
\|(\text { ?id.freeinds }- \text { ?b.freeinds }) \|=\text { eval(?expr) }
$$

where eval(?expr) evaluates the expression by substituting all dMVs and template items by the corresponding cardinalities.

### 3.1.4 IW - Access methods

Once the data is verified to be consistent with the model, one may access the model's data related information, through the IW's access methods.

- dimSize(?dim) -> ?size,
- constVal(?icons) $->$ ?val,
- upperBnd(?ivar) $->$ ?bnd and
- lowerBnd(?ivar) $->$ ?bnd.

For accessing the data through IW, there two possibilities, the first one is to use the global names (even the ones automatically generated for $\operatorname{LfP}^{\mathcal{L}}$ s) and the second one is to use an internal parameter type identifier or a dMV (for dimensions). For example, we may want to know the size of a dimension named d1 or the size of a dMV of a certain structure.

Let's define how both type of access work for each argument of the previously mentioned access methods,

- For ?dim
- case 1 (using global names):
* When we ask for a dimension defined as d_dimension(?d), then ?dim = ?d
* When the dimension is defined solely as d_dimension(?d, ? pll), if ? pll . freeinds $=\emptyset$ then ?dim = ?d, otherwise, ?dim $=$ ?d $\left(\mathrm{fi}_{1}, \ldots, \mathrm{fi}_{n}\right)$ where ( $\left.\mathrm{fi}_{1}, \ldots, \mathrm{fi}_{n}\right) \in$ ? pll.freeinds ${ }^{\xi}$.
- case 2 (using dMV names): Given ?P:?C and ?C::d_LeafProblem_C $\left(\operatorname{LfP}\right.$ or $\left.\operatorname{LfP}^{\mathcal{L}}\right)$ if we want to know the size of a dMV ?d $\in$ ?C.dim_var for that ?P then
* if ?P. freeinds $=\emptyset$ or ?C[not local] then ? $\operatorname{dim}=? \mathrm{P}(? \mathrm{~d})$.
* otherwise, ? dim $=? \mathrm{P}\left(\right.$ ? $\mathrm{d}^{\left(\mathrm{fi}_{1}, \ldots, \mathrm{fi}_{n}\right)}$ ) where $\left(\mathrm{fi}_{1}, \ldots, \mathrm{fi}_{n}\right) \in$ ?pll.freeinds $\xi$.
- For ?icons
- case 1 (using global names):
* if we ask for an indexed constant of the form ?c $\left(\mathrm{i}_{1}, \ldots, \mathrm{i}_{n}\right)$ then ?icons $=? c\left(\mathrm{i}_{1}^{v}, \ldots, \mathrm{i}_{n}^{v}\right)$, where $\left(\mathrm{i}_{1}^{v}, \ldots, \mathrm{i}_{n}^{v}\right) \in\left(\mathrm{i}_{1}, \ldots, \mathrm{i}_{n}\right)^{\xi}$.
* if the indexed constant has the form ?c $(? \mathrm{pII})\left(\mathrm{i}_{1}, \ldots, \mathrm{i}_{n}\right)$ then ?icons $=? \mathrm{c}(? \mathrm{pII})\left(\mathrm{i}_{1}^{v}, \ldots, \mathrm{i}_{n}^{v}\right)$, where $\left(\mathrm{i}_{1}^{v}, \ldots, \mathrm{i}_{n}^{v}\right) \in\left(\mathrm{i}_{1}, \ldots, \mathrm{i}_{n}\right)^{\xi}$ and ? pll :d_LeafProblem_C[local].
- case 2 (using type parameter names): Given ?P:?C and ?C::d_LeafProblem_C (LfP or LfP ${ }^{\mathcal{L}}$ ) if we want to know the value of a type parameter ? tid such that (? tid = ?type) $\in$ ?C.args then
* if ?P. freeinds $=\emptyset$ and ?type is not a vector type then ? icons = ?P(?tid).
* if ?P. freeinds $=\emptyset$ and ?type $=$ d_vector(?K, ?S) then ?icons $=? \mathrm{P}\left(?\right.$ tid $\left.\left(\mathrm{i}_{1}^{v}, \ldots, \mathrm{i}_{|? \mathrm{~S}|}^{v}\right)\right)$, where $\left(i_{1}^{v}, \ldots, i_{\mid ? S}^{v}\right) \in ? S^{\xi}$.
* if ?P. freeinds $\neq \emptyset$ and ?type is not a vector type then ?icons $=? \mathrm{P}\left(?\right.$ tid $\left.\left(\mathrm{fi}_{1}, \ldots, \mathrm{fi}_{n}\right)\right)$, where ( $\mathrm{fi}_{1}, \ldots$. fi ${ }_{n}$ ) $\in$ ? P.freeinds ${ }^{\xi}$.
* if ?P. freeinds $\neq \emptyset$ and ?type $=d_{\text {_vector(? }}$ K, ?S $)$ then ?icons $=? \mathrm{P}\left(? \operatorname{tid}\left(\mathrm{fi}_{1}, \ldots, \mathrm{fi}_{n}, \mathrm{i}_{1}^{v}, \ldots, \mathrm{i}_{|? \mathrm{~S}|}^{v}\right)\right)$, where $\left(\mathrm{i}_{1}^{v}, \ldots, \mathrm{i}_{\mid{ }^{v}}^{v} \mathrm{~S} \mid\right) \in$ ? $^{\xi}$ and $\left(\mathrm{fi}_{1}, \ldots ., \mathrm{fi}_{n}\right) \in$ ?P.freeinds ${ }^{\xi}$.
- For ? ivar is the same as ?icons but instead of constant identifier we use variable identifiers.

The IW's access methods will rely on the DH's access methods. The IW will transform each access method's argument to be used by the DH , according to the specification given in $\S 3.1 .2$. Therefore all global names will be converted to the corresponding dMV name (in case of a dimension) or the type parameter name (in case of a constant or variable).

When manipulating $\operatorname{LfP}^{\mathcal{L}}$ s elements, the IW will also try to retrieve the corresponding DH , based on the $\operatorname{LfP}^{\mathcal{L}}$ free indices. If the $\mathrm{LfP}^{\mathcal{L}}$ has more than one DH , the IW will linearize the instanced free indices to a value $i$ and using $i$-th DH will call the corresponding DH access method. However, if the amount of DHs is 1 , then the IW will pass to that DH the $\zeta\left(\mathrm{fi}_{1}, \ldots, \mathrm{fi}_{n}\right)$ so that the DH searches and returns the proper dimension size (or constant value or variable bound).

### 3.2 Structured Instance Generation

Besides accessing the data of a certain model, the IW enables I-DARE to generate a structured instance (SI). The SI will be composed of a main XML file called Meta Data and a set of external references (see Figure 3.1). As will be seen later, the external references may be to data files or remote data files. There is no restriction in how a external reference is designed, it only requires that the corresponding DH (and eventually the solver) knows how to deal with it.


Figure 3.1: Structured Instance

### 3.2.1 Meta Data File

The meta data file will be a XML containing all the necessary information about the structure and the external references. The meta data file will be structured like the internal model, starting from the root node and unfolding towards the leaf problems. To show how the meta data file is formed we will use a bottom up fashion, from the leafs to the root.

## Leaf Problems

A leaf problem is represented using the following XML tag,
Listing 3.8: LEAF Node general form

```
<LEAF type="class">
    <VARS>
        <V n=" full_name" />
    </VARS>
    <DATA size="reps">
    </DATA>
</LEAF>
```

The main tag contains a parameter named type which must be a ?C::d_LeafProblem_C. Observe there is no need to store the name of the leaf problem we are writing, we just need the its class name. Structures' names play an important part during all pre-solving phases, but once the model+data is adequately validated there is no need of them, anymore. The solvers only need the to know which class the structure is, not its name in the model.

Immediately afterwards, there is a tag named VARS that contains a sequence of tags $V$ representing the leaf problem variable names. Each $\vee$ tag has a n parameter to which we assign the full name of an indexed variable belonging to the leaf problem we are writing. The full name is a term ? id $\left(\mathrm{i}_{1}^{v}, \ldots, \mathrm{i}_{n}^{v}\right)$, where ? id :d_var and $\left(\mathrm{i}_{1}^{v}, \ldots, \mathrm{i}_{n}^{v}\right) \in\left(\mathrm{i}_{1}, \ldots, \mathrm{i}_{n}\right)^{\xi}$, assuming $\mid$ ?id.dims $\mid=n$.

For example if we have a variable named $\times$ with dimensions d 1 and d 2 , assuming $\mathrm{d} 1=\{0,1\}$ and $\mathrm{d} 2=\{0,1,2\}$ and that d 1 is a free dimension for the leaf problem we are trying to represent, then since d 1 is a free dimension we will write the leaf problem two times for $\mathrm{d} 1=0$ and for $\mathrm{d} 1=1$. Taking $\mathrm{d} 1=0$ then it will be generated one V tag for each of the following $\times(0,0), \times(0,1)$ and $\times(0,2)$.

The DATA tag is defined depending on whether the leaf problem is local or not. Given an instance wrapper ?IW, ?P:?C, ?C::d_LeafProblem_C, for each ( $\mathrm{fi}_{1}, \ldots, \mathrm{fi}_{n}$ ) $\in$ ?P.freeinds ${ }^{\xi}$ we define a dictionary ?ID $=\left\{\right.$ ?P.freeinds $[1]=\mathrm{fi}_{1}, \ldots$, ?P. freeinds $\left.[n]=\mathrm{fi}_{n}\right\}$ then if

- ?C[not local] - let ?gDH be the DH defined for global, then
?gDH[dataTag(?IW, ?P, ?ID) $->$ [?xmltags, ?vars]] defines the XML Nodes to be added to DATA (?xmitag) and a set of variables to be added to VARS (?vars).
- ?C[local] - let ?IDH be the DH obtained from ?IW.local(?P) $\left[\zeta\left(\mathrm{fi}_{1}, \ldots, \mathrm{fi}_{n}\right)\right]$, then ?IDH[dataTag(?IW, ?P, ?ID) -> [?xmltags, ?vars]] defines the XML Nodes to be added to DATA (?xmltag) and a set of variables to be added to VARS (?vars).

The parameter size of tag DATA will be equal to |?P. freeinds ${ }^{\xi} \mid$ (i.e. the amount of replications of the leaf problem depends on the amount of instanced free indices).

For example,
$\frac{\text { piece of }}{\text { ex2 : d_LP_MPS_C }}$
[
args $->$ [
[
$x=x(i 1 \$ d 1)$
$c=c(i 1 \$ d 1)$
$A=A(j 1 \$ d 2, i 1 \$ d 1)$,
$\mathrm{b}=\mathrm{b}(\mathrm{j} 1 \mathrm{\$ d} 2)$
[fi]
$]^{[f i]}$
].

Assuming the MPS file "data.mps" contains 3 variables and defines 2 RHS vectors, and that fi has size 2, then the meta data file generation algorithm will create the following LEAF tag,

```
<LEAF type="d_LP_MPS_C">
    <VARS>
        <V n="x(ex2)(0,0)" />
        <V n="x(ex2)(0,1)"/>
        <V n=" x (ex2)(0,2)" />
        <V n=" x (ex2) (1,0)" />
        <V n=" }\times(e\times2)(1,1)"/
        <V n=" x (ex2)(1,2)" />
    </VARS>
    <DATA size="2">
        <MPSFile name="data.xml" irhs="0" />
        <MPSFile name=" data.xml" irhs="1"/>
    </DATA>
</LEAF>
```

Observe that the dataTag method of the d_MPSDataHandler generates a series of MPSFile tags, depending on the amount of RHS vectors and free indices of the $\mathrm{LfP}^{\mathcal{L}}$. Each MPSFile tag will contain the reference to the MPS data file and which RHS vector is used. The variables have two indices, because a $\operatorname{LfP}^{\mathcal{L}}$ property adds at the beginning of its dimensions the $\operatorname{LfP}^{\mathcal{L}}$,s free index dimension (see §2.4.1). Note that using the internal DH replication one may economize the space used to store the final structured instance. In this case the main problem structure is shared and the RHS vector change.

Another way of obtaining a similar result is by having two MPS data files with same main structure only changing the RHS vector.

```
enhanced instance declaration
mpsdh1 : d_MPSDataHandler
mps
    MPSfile -> "data1.mps"
].
mpsdh2 : d_MPSDataHandler
[
    MPSfile -> "data2.mps"
].
iw : d_InstanceWrapper
[
    local(ex2) -> [mpsdh1, mpsdh2]
].
```

```
LEAF XML Node
<LEAF type="d_LP_MPS_C">
    <VARS>
        \(<\mathrm{V} n=" \times(\mathrm{e} \times 2)(0,0) " />\)
        \(<\mathrm{V} n=" \times(\mathrm{e} \times 2)(0,1) " />\)
        \(<\mathrm{V} n=" \times(e \times 2)(0,2) " />\)
        \(<\mathrm{V} n=" \times(\mathrm{e} \times 2)(1,0) " />\)
        \(<\mathrm{V} n=" \times(\mathrm{ex} 2)(1,1) " />\)
        \(<\mathrm{V} n=" \times(\mathrm{e} \times 2)(1,2) " />\)
    </VARS>
    <DATA size=" \(2 ">\)
        <MPSFile name="data1.xml" irhs=" 0 " />
        <MPSFile name="data2.xml" irhs="0"/>
    </DATA
```

In this case the replication is not done inside the d_MPSDataHandler. The IW will call first the dataTag method of mpddh1 and then the one from mpsdh2. Each one of the previous calls returns one MPSFile tag, pointing to the corresponding MPS file and to the only RHS vector inside of it.

Since the difference between data1.mps and data2.mps is only the RHS vector, it is recommendable to use only one file and thus only one DH . The feature of internal replication depends on how one implements the DH (see §3.1.2).

To show how the same mechanism works for a $\operatorname{LfP}$ instead of a $\operatorname{LfP}{ }^{\mathcal{L}}$, we will use the following example,

```
data.xml
< <DIMS> 
    enhanced instance declaration
gdh : d_XMLDataHandler
[
    xmlfile -> "data.xml"
].
iw : d_InstanceWrapper
iw
    global -> gdh
].
```


## piece of a model

```
d_dimension(d1).
```

d_dimension(d1).
d_dimension(d2).
d_dimension(d2).
d_index(i, d1).
d_index(i, d1).
d_index(j, d2).
d_index(j, d2).
a:d_constant
a:d_constant
a:d_property
a:d_property
dims -> [d2, d1]
dims -> [d2, d1]
].
].
x:d_var.
x:d_var.
x:d_property
x:d_property
[
[
dims }->\mathrm{ [d1],
dims }->\mathrm{ [d1],
upper -> 1,
upper -> 1,
lopper -> 1,
lopper -> 1,
].
].
ex1 : d_LP_C
ex1 : d_LP_C
[ args -> [
[ args -> [
args -> [ }\quadx=$(x(i),[i])
    args -> [ }\quadx=$(x(i),[i])
x=$(x(i),[i])
            x=$(x(i),[i])
A=$(a(j,i), [i,j])
            A=$(a(j,i), [i,j])
A=$(a(j,i}),[i,j]), l ) [j])
            A=$(a(j,i}),[i,j]), l ) [j])
rels = \$('<', [j]),
rels = \$('<', [j]),
dir =',min'
dir =',min'
]
]
].

```
].
```

To write this LfP the Meta Data generator will use the DH subscribed to global (d_XMLDataHandler). The dimensions and constants will be extracted from the global XML data file, and using them the system will create the following LEAF XML node,

LEAF XML Node

```
<LEAF type="d_LP_C">
    <VARS>
        <V n=" x (0)"/>
        <V n="x(1)" />
        <V n="x(2)" />
    </VARS>
    <DATA size="1">
        <ARGS>
            <VECTOR name="x" type="d_var" size=" 3">
                <V i="0" v="x(0)" up ="1" 1o = "0" />
                <V i="1" v="x(1)" up = "1" lo = "0"/>
                <V i="2" v="x(2)" up ="1" Io = "0" />
            </VECTOR>
            <VECTOR name=" c" type="d_constant" size=" 3">
                <V i="0" v="1"/>
                <V i=" 1" v="1" >
                <V i=" 2" v="1" />
            </VECTOR>
            <VECTOR name="A" type="d_constant" size=" 3,3">
                <V i="0,0" v=" 0.2" />
                <V i=" 1,0" v=" 0.6"/>
                <v i=" 1,1", v="-2" />
                <V i=" 2,1" v="3"/>
                <V i="1,2" v="2">
            </VECTOR>
            <VECTOR name="b" type=" d_constant" size=" 3">
                <V i="1" v="1" />
                <V i="2" v="1"/>
            </VECTOR>
            <VECTOR name=" rels" type=" d_rel" size=" 3">
                <V i="0" v="&lt;"/>
                <V i="1", v="&lt;">>
                <V i="2" v="&lt;"/>
            </VECTOR>
            <DIRECTION name="dir" value=" min" />
        </ARGS>
```


## </DATA > <br> </LEAF $>$

The dataTag method of d_XMLDataHandler returns a series XML Nodes named ARGS (depending on the LfP's free indices). ARGS will contain one XML Node for each LfP's argument. In the previous example, note that the vectors contain one tag for each non-zero value.

## Blocks

In the enhanced instance there must be a block representation, in order to export to the solver the model's structure composition. A block is written inside the meta data file in the following way,

Listing 3.9: BLOCK Node general Form

```
<BLOCK type="tn">
    <SUBS>
        <SUB name = "id", size = "rep">
        </SUB>
    <SUBS/>
</Block>
```

Assuming we are trying to write a block ?B:?C[not abstract], where ?C::d_Block_C, then a tag BLOCK will be created with type $=$ "? C" for each free index of ?B. For each substructure ?B.subs $[i]$ the system will create a tag SUB, with size $=$ " $\|$ ? B.subs $[i]$. freeinds - ?B.freeinds $\| "$ and name $=$ "?C.ids $[i] "$. Note that size represents the amount of replications of the substructure inside ?B. Hence, for each instance of the free indices, the meta generation algorithm will be called to create the tags of the substructures.

For example, if we have the following block inside a model,

```
milpB : d_B_MILP_C
[
    subs -> [ex3, ic ],
    subVP ->> [[y,x], [y]]
].
```

Then the meta generation algorithm will create this BLOCK tag,
BLOCK XML Node

```
<BLOCK type="d_B_MILP_C">
    <SUBS>
        <SUB name = "|p", size = "1">
            <-- LEAF Node generated from ex3 -->
        </SUB>
        <SUB name = "ic", size = "1">
            <-- LEAF Node generated from ic -->
        </SUB>
    <SUBS/>
</Block>
```

Assume now the ex 3 substructure has one free index of size 3 . Then the BLOCK tag will look like,

## BLOCK XML Node

```
<BLOCK type="d_B_MILP_C">
    <SUBS>
        <SUB name = "Ip", size = "3">
            <-- LEAF Node generated from ex3 for free index instance 0 -->
            \ll - ~ L E A F ~ N o d e ~ g e n e r a t e d ~ f r o m ~ e x 3 ~ f o r ~ f r e e ~ i n d e x ~ i n s t a n c e ~ 1 ~ - - > ~
            <-- LEAF Node generated from ex3 for free index instance 2 -->
        </SUB>
```

```
    <SUB name = "ic", size = "1">
        <-- LEAF Node generated from ic -->
        </SUB>
    <SUBS/>
</Block>
```

Observe that ex3 will be replicated for each instance of its free index, since the block's free indices is $\emptyset$. If milpB had defined the freel method, including the same free index of ex3 as its own free index, then the system would have generated 3 BLOCK tags.

## Formulation

Finally there is an outer tag that contains the whole formulation, defined as follows,

```
<FORMULATION name=" form_name">
    <- Component tree, starting with the root component -->
<FORMULATION/>
```

The FORMULATION tag has a name parameter, which must be the formulation identifier, and only one child tag that could be a collection of BLOCK tags or LEAF tags depending on the formulation's root. Note that the collection size directly depends on \|root.freeinds $\|$ (i.e. amount of root's free index instances).

### 3.3 Discussion

In this chapter we defined how the actual problem's data is integrated in a harmonic way with the IM of the problem. I-DARE(ei) defines an extension to the formulation, called Instance Wrapper (IW), that using the previously registered Data Handlers (DH), enables the user to query the extended model.

The DHs allow the system to treat a wide variety of data formats, offering the system a transparent interface to verify and access the data. At the same time the IW takes the abstraction to a higher level, by making transparent the DHs it will use to retrieve the data. The IW allows to query the model's data by using global names, dMV and even type identifiers, automatically detecting which DH must be used.

The IW has another functionality (other than querying the data), it generates the structured instance, to be passed to the solvers (together with the Solvers' Tree and the Configurations (cf. $4)$ ). The SI is generated in a structured way, separating the meta data file (with the structural information) from the local data (that may be represented in any format and support). This way the data may remain decentralized and may favor the distributed application of solvers.

## Chapter 4

## I-DARE(solve) - solving the model


#### Abstract

This chapter will describe the solving module of I-DARE (I-DARE(solve)). This module is responsible for actually performing the solution approach on a formulation + instance, collecting and presenting the result to the core system. Here we will define the concept of Solvers' Tree as the description of the (set of) numerical solver(s) that has(have) been selected by the search process as the most appropriate for the given SI; this is not just a solver but, in general, a structured collection of solvers, some of which using others to cope with specialized structures. The I-DARE(solve) module will orchestrate the actual solution process relying on the available set of solvers plus their configurations, i.e., implementations of the general solver interface that allows to plug specific solvers to the I-DARE system.


### 4.1 Structured Modeling and Solving Methods

The first stage for solving any problem, is to obtain an informal specification of it. Generally these informal specifications are given by a person who is not a specialist in MO. This specification is then refined and finally taken by an expert and transformed into a mathematical model. Usually there is a lot of difference between the final model and the informal specification. Moreover, in the transformation process structural information about the problem may be lost.

The following four sections will show examples of different structures that are implicitly present in problems. In addition, it will be shown that specific algorithms (or combination of them) can be applied for tackling the problem and obtain a solution in a reasonable amount of time, once the structure is recognized.

### 4.1.1 Some initial formulation

Assume the expression $x y$ appears somewhere in a model, where $x \in[0, u] \subseteq \mathbb{R}$ and $y \in\{0,1\}$, makes the model nonlinear. Observe that $x y=0$ iff $y=0$ or $x=0$.

Therefore, one could make another equivalent formulation of that expression, substituting $x y$ by $x$, eliminating the bounding constraint $x \in[0, u] \subseteq \mathbb{R}$ and adding the following

$$
\begin{array}{r}
0 \leq x \leq y u \\
y \in\{0,1\} \\
x \in \mathbb{R}
\end{array}
$$

The obtained formulation is a MILP one, and for this reason it would typically be preferred by the vast majority of experts, who 1) are more accustomed with MILP models, and 2) think that MILP formulations are more efficient than nonlinear ones. But the unexperienced user, who
does not know about reformulation tricks between nonlinear models and linear ones, and does not want to be confronted with the algorithmic difference between a MILP model and a nonlinear one, will most likely find the "xy" formulation much more natural, compact and easy to understand.

Furthermore, let us consider the binary constraint $y \in\{0,1\}$. A binary constraint is typically seen as an "atomic" object by a person who is specialist in MILP modeling, but in a very nonlinear problem where this is the "only" discrete part, it may be more reasonable to "nonlinearize" it rather than working with the discrete object.

Therefore the previous binary constraint could be reformulated as the nonlinear (nonconvex) constraint $y(1-y)=0$, where $y \in \mathbb{R}$. Alternatively this could be also reformulated as $y(1-y) \leq 0$, where $y \in[0,1] \subset \mathbb{R}$. Both reformulations enforce that $y$ must be binary, since the zeros of the function are exactly 0 and 1 , see figure 4.1.


Figure 4.1: Representation of $y(1-y)$ function

### 4.1.2 Knapsack constraints

Certain products should be loaded on a truck of capacity $c$. Let $S$ be the set of products and $w_{l}$ the weight of an unit of $l(l \in S)$, then must be enforced that $\sum_{l \in S} w_{l} x_{l} \leq c$, where $x_{l}$ is the amount of units of $l$ that will be loaded on the truck.

This problem can be modeled by knapsack packing constraints. Assume ksPack( $\mathrm{x}, \mathrm{w}, \mathrm{c}$ ) is a description of a knapsack packing constraint, where $\times$ are the element amounts to be loaded, w are the weights for unit of each element and c is the capacity in weight of the knapsack. Constructs like this one can be use inside a modeling language for expressing in a concise and structured way knapsack packing constraints.

Additionally, knapsack packing constraints can be also modeled using a MILP approach,

$$
\begin{array}{r}
\sum_{l \in S} w_{l} x_{l} \leq c  \tag{4.1.1}\\
x_{l} \in \mathbb{N}_{0}
\end{array}
$$

In this case the complexity of the constraint and the MILP model seems similar. However, acknowledging that one is using the knapsack packing constraint allows to apply specific bound propagation. The domain of $x_{l}$ could be represented as a set $\left\{L_{l}, L_{l}+1, \ldots, U_{l}-1, U_{l}\right\}$ of consecutive integers. Then bound propagation will tighten the upper bounds $(U)$.

If there is a knapsack packing constraint, then for all $l \in S$

$$
x_{l} \leq \frac{c-\sum_{\substack{l^{\prime} \in S \\ l^{\prime} \neq l}}\left(w_{l^{\prime}} x_{l^{\prime}}\right)}{w_{l}} \leq \frac{c-\sum_{\substack{l^{\prime} \in S \\ l^{\prime} \neq l}}\left(w_{l^{\prime}} L_{l^{\prime}}\right)}{w_{l}}
$$

the second inequality is due to $x_{l} \geq L_{l}$ and $w_{l}>0, \forall l \in S$. Then the upper bounds can be recalculated as follows,

$$
\begin{equation*}
U_{l}^{\prime}=\min \left\{U_{l},\left[\frac{c-\sum_{\substack{l^{\prime} \in S \\ l^{\prime} \neq l}}\left(w_{l^{\prime}} L_{l^{\prime}}\right)}{w_{l}}\right\rfloor\right\} \tag{4.1.2}
\end{equation*}
$$

where $\lfloor\alpha\rfloor$ is $\alpha$ rounded down.
Assume now that instead of enforced a capacity constraint, a demand must be satisfied. Then the constraint changes in the following way, $\sum_{l \in S} w_{l} x_{l} \geq d$.

This problem now can be modeled by knapsack covering constraint (ksCover(x, w, d)), where x are the element amounts to be loaded, w are the weights for unit of each element and d is the demand in weight of the knapsack.

MILP could be applied for modeling knapsack covering constraints,

$$
\begin{align*}
\sum_{l \in S} w_{l} x_{l} & \geq d  \tag{4.1.3}\\
x_{l} & \in \mathbb{N}_{0}
\end{align*}
$$

In this case, the knapsack covering constraint helps to tighten lower bounds in the following way,

$$
\begin{equation*}
L_{l}^{\prime}=\max \left\{L_{l},\left\lceil\frac{d-\sum_{\substack{l^{\prime} \in S \\ l^{\prime} \neq l}}\left(w_{l^{\prime}} U_{l^{\prime}}\right)}{w_{l}}\right\rceil\right\} \tag{4.1.4}
\end{equation*}
$$

where $\lceil\alpha\rceil$ is $\alpha$ rounded up.
In the presence of both knapsack packing and covering constraints in the same problem involving common variables, successive applications of (4.1.2) and (4.1.4), may help to tighten the bounds of the variable domains. This bound propagation can be combined with linear programming relaxation, which removes the integrality constraints and a branch and bound technique. Based on generated cuts (from bound propagation) branch and bound decides whether a relaxed solution is acceptable or not [94].

## 0-1 Knapsack constraints

A particular case of a knapsack problem is when, instead of loading a certain amount $x_{l}$, one must decide whether $l$ is going to be loaded or not i.e. $x_{l} \in\{0,1\}$. This transformation makes the previous knapsack problems into binary ones, also known as $0-1$ knapsack problem.

0-1 Knapsack constraints can also be modeled by $\operatorname{ksPack}(\mathrm{x}, \mathrm{w}, \mathrm{c})$ and $\mathrm{ksCover}(\mathrm{x}, \mathrm{w}, \mathrm{d})$, considering x are binary variables. However, for making clear the difference between the 0-1 knapsack constraint structure and the non binary one, $\operatorname{ks01Pack}(x, w, c)$ and $k s 01 \operatorname{Cover}(x, w, d)$ will be used in the binary case.

Recognizing explicitly this structure, gives the possibility of applying specific techniques. For instance, cuts could be derived from cover inequalities and then a lifting process could be applied to strengthen these inequalities [26].

Define a cover for ks01Pack(x, w, c) to be an index set $J \subseteq\{1, \ldots,|S|\}$ for which $\sum_{j \in J} w_{j}>c$. A cover is minimal if no proper subset is a cover. If $J$ is a cover, the following is a cover inequality,

$$
\begin{equation*}
\sum_{j \in J} x_{j} \leq|J|-1 \tag{4.1.5}
\end{equation*}
$$

Only minimal covers will be considered.
Cover inequality (4.1.5) generates a cutting plane, which could be stronger if lifted to higher dimensional space, by adding new terms as follows,

$$
\begin{equation*}
\sum_{j \in J} x_{j}+\sum_{j \in J^{\prime}} \pi_{j} x_{j} \leq|J|-1 \tag{4.1.6}
\end{equation*}
$$

where $J \cap J^{\prime}=\emptyset$ and every new $\pi_{k}$ with $k \notin L=J \cup J^{\prime}$ is calculated as,

$$
\begin{equation*}
\pi_{k}=|J|-1-\max _{\substack{x_{j} \in\{0,1\} \\ \text { for } j \in L}}\left\{\sum_{j \in J} x_{j}+\sum_{j \in J^{\prime}} \pi_{j} x_{j} \mid \sum_{j \in L} w_{j} x_{j} \leq c-w_{k}\right\} \tag{4.1.7}
\end{equation*}
$$

One can also make a reformulation of $\operatorname{ks01Pack}(x, w, c)$ as a directed acyclic graph (DAG). Let $G(N, E)$ be a DAG, where $N$ and $E \subset N \times N$ are constructed sequentially from 1 to $|S|$. Each node will be labeled with $n^{l, \beta}$, where $l \in S \cup\{0\}$ and $0 \leq \beta \leq c$. Let $f: E \longrightarrow \mathbb{R}$ be the weights of the arcs in $E$. Assume elements in $S$ are consecutive numbers from 1 to $|S|$. Let $n^{0,0}$ be an initial node in $N$, then for each $n^{l, \beta} \in N$ the following must be done,

- $\operatorname{add} N=N \cup\left\{n^{l+1, \beta}\right\}, E=E \cup\left\{\left\langle n^{l, \beta}, n^{l+1, \beta}\right\rangle\right\}$ and $f\left(\left\langle n^{l, \beta}, n^{l+1, \beta}\right\rangle\right)=0$ if $l+1 \leq|S|$,
- add $N=N \cup\left\{n^{l+1, \beta+w_{l+1}}\right\}, E=E \cup\left\{\left\langle n^{l, \beta}, n^{\left.l+1, \beta+w_{l+1}\right\rangle}\right\rangle\right.$ and $f\left(\left\langle n^{l, \beta}, n^{l+1, \beta+w_{l+1}}\right\rangle\right)=-1$ if $l+1 \leq|S|$ and $\beta+w_{l+1} \leq c$
this process will be repeated for each new addition to $N$ until no further addition can be made. Then, finally $n^{|S|+1, c+1}$ will be added to $N$ and for each $n^{|S|, \beta},\left\langle n^{|S|, \beta}, n^{|S|+1, c+1}\right\rangle$ will be added to $E$, with $f\left(\left\langle n^{|S|, \beta}, n^{|S|+1, c+1}\right\rangle\right)=0$.

Let $s^{l, \beta}$ be a net supply associated with each node $n^{l, \beta}$, such as,

- $s^{0,0}=1$,
- $s^{|S|+1, c+1}=-1$ and
- for any other, $s^{l, \beta}=0$
and let $y^{\left\langle n^{l, \beta}, n^{l^{\prime}, \beta^{\prime}}\right\rangle}$ be a real-valued flow variables associated with every arc.
Then the following constraints,

$$
\begin{align*}
\sum_{\left\langle n^{l, \beta}, n^{l^{\prime}, \beta^{\prime}}\right\rangle \in E} y^{\left\langle n^{l, \beta}, n^{n^{\prime}, \beta^{\prime}}\right\rangle}- & \sum_{\left\langle n^{\left.l^{\prime}, \beta^{\prime}, n^{l, \beta}\right\rangle \in E}\right.} y^{\left\langle n^{\left.l^{\prime}, \beta^{\prime}, n^{l, \beta}\right\rangle}\right.}=s^{\left\langle n^{l, \beta}, n^{l^{\prime}, \beta^{\prime}}\right\rangle} \text {, for all } n^{l, \beta} \in N  \tag{4.1.8}\\
& \sum_{\left\langle n^{l, \beta}, n^{l, \beta^{\prime}}\right\rangle \in E} y^{\left\langle n^{l, \beta}, n^{l, \beta^{\prime}}\right\rangle}=1, \text { for all } l \in S \tag{4.1.9}
\end{align*}
$$

ensures flow conservation in $G$ using a unique path.
Therefore if

$$
x_{l}= \begin{cases}1 & \text { if } \sum_{\left\langle n^{l-1, \beta}, n^{l, \beta^{\prime}}\right\rangle \in E} y^{\left\langle n^{l-1, \beta}, n^{l, \beta^{\prime}}\right\rangle} f\left(n^{l-1, \beta}, n^{l, \beta^{\prime}}\right)=-1 \\ 0 & \text { otherwise }\end{cases}
$$

then a feasible solution of the flow problem allows to construct a feasible solution of ks01Pack( $\mathrm{x}, \mathrm{w}, \mathrm{c}$ ). In fact, constraints (4.1.8) and (4.1.9) define a convex hull of integer feasible solutions to the problem.

## Semi-assignment constraints

Semi-assignment constraints are a very particular case of 0-1 knapsack constraints, which ensures that a sum of binary variables is exactly 1 ,

$$
\begin{align*}
\sum_{j} y_{i j} & =1 \forall i  \tag{4.1.10}\\
y_{i j} & \in\{0,1\} \forall i, j
\end{align*}
$$

it means that one thing can be assigned to a set of other things, but it cannot be assigned to more than one.

The semi-assignment constraints have the integrality property, i.e. there is no tricky transformation needed for having the convex hull of the integer solutions. Putting together two semiassignment constraints

$$
\begin{align*}
\sum_{i} y_{i j} & =1 \forall j  \tag{4.1.11}\\
\sum_{j} y_{i j} & =1 \forall i  \tag{4.1.12}\\
y_{i j} & \in\{0,1\} \forall i, j
\end{align*}
$$

an assignment problem is obtained, which can be solved in polynomial time (it is a special case of the min-cost flow problem, which can be solved also in polynomial time even for integer variables if all data is integer).

Therefore there will be a specific identifier for the semi-assignment constraint, semiassign ( $\times[i, j]$ ), where $\times[i, j]$ is a matrix of variables ( $i$ are the rows and $j$ are the columns). For each row there will be a constraint which will enforce that the sum of every $\times$ in that row is exactly equal to 1 .

There could also be defined a semi-assignment structure which transforms the equality into an inequality, i.e. instead, of being exactly 1 , the sum can be $\leq 1$. For this case the following signature can be used, semiassignLEQ( $\times[\mathrm{i}, \mathrm{j}]$ ) (LEQ meaning Less or EQual).

## Independent set constraints

When we are in presence of a set of 0-1 knapsack packing constraints, which ensure that the sum of certain pairs of variables have to be less or equal than 1 , we are dealing with another specific structure, called independent set constraints.

$$
\begin{align*}
y_{i}+y_{i^{\prime}} & \leq 1 \forall i, i^{\prime} \text { that satisfy a certain condition }  \tag{4.1.13}\\
y_{i} & \in\{0,1\} \forall i
\end{align*}
$$

finding a feasible solution on the previous constraints is equivalent to find an independent set on a graph $G(V, E)$ where each $y_{i}$ defines a node in $V$ and a constraint between $y_{i}$ and $y_{i^{\prime}}$ defines an arc in $E$. A set $I \subseteq V$ is considered independent iff $\langle v, w\rangle \notin E$ for all $v, w \in I$. An independent set of nodes in $G$ is a feasible solution for the independent set constraints.

It is also important to underline the importance of specifying an explicit identifier for the independent set constraints structure, indset ( x , cond), where $\times$ is a vector of binary variables and cond is a logical proposition for building the independent set constraints (arcs in $G$ ) between the members of x . An independent set constraint will relate two variables of x , one referred as $x_{i}$ and the second as $x_{i^{\prime}}$, then in cond for making reference to the indices of the second variable one shall use ", ". The usage of a logical proposition in cond to construct the graph might bring an unwanted complexity while evaluating each possible arc, therefore in real implementations may be used simple logical formulas or a set of arcs instead of a logical condition.

Finding independent sets in a graph is a widely studied problem, for which, despite the efforts, there is no polynomial time algorithm (is a NP-Complete problem). But there are many algorithms that have an average good performance like [148, 139].

The complement of $G$ will be the graph $\bar{G}(V, \bar{E})$, where $\bar{E}=\{\langle v, w\rangle \mid v, w \in V, v \neq w \operatorname{and}\langle v, w\rangle \notin$ $E\} . I \subseteq V$ is an independent set of $G$ iff $I$ is a clique of $\bar{G}$. Therefore all the algorithms developed for finding cliques in a graph can be applied to find independent sets on its complement [150, 158].

### 4.1.3 Logical Formula

Suppose that while making an informal specification of a problem, a logical proposition like the following is stated, "there is an empty truck or there is a partially loaded truck and it goes to the same destination I want to go". Then the truth value of that proposition can be codified by the following equation,

$$
\begin{equation*}
d=a \vee(b \wedge c) \tag{4.1.14}
\end{equation*}
$$

where,
a represents the proposition there is an empty truck,
$b$ represents the proposition there is a partially loaded truck and
$c$ represents the proposition it goes to the same destination I want to go.
$d$ represents the truth value of the proposition there is an empty truck or there is a partially loaded truck and it goes to the same destination I want to go.

Equation (4.1.14) can be transformed in the following linear system,

$$
\begin{aligned}
c^{\prime} & \leq b \\
c^{\prime} & \leq c \\
c^{\prime} & \geq b+c-1 \\
d & \geq a \\
d & \geq c^{\prime} \\
d & \leq a+c^{\prime}
\end{aligned}
$$

In the moment (4.1.14) is transformed in a linear system all the structural information is lost. Assuming these linear inequalities are part of a larger model, then a solver will never be aware that they where a logical formula, where satisfiability algorithms could be applied, like, Davis-PutnamLoveland (DPL) algorithm; also inference methods like, unit resolution [94] or parallel resolution [94]; as well as convex hull relaxations for logical formulas [91].

### 4.1.4 Disjunctive Scheduling

A Distribution Center with one truck platform must decide when to attend $n$ trucks in such a way that they do not overlap. Each truck has a starting time $e_{i} \in[L, U] \subseteq \mathbb{R}$ and a processing time $p_{i}$, which defines how long the truck will remain at the platform; $L$ and $U$ are the opening and closing time of the distribution center and $i \in[1 . . n]$. Which values are assigned to $e_{i}$ in such a way that $e_{j} \geq e_{i}+p_{i}$ or $e_{i} \geq e_{j}+p_{j}, \forall i, j \in[1 . . n]$ must be decided, meaning that there must not be overlapping at the platform.

The previous problem could be modeled using a CP global constraint named disjunctive scheduling constraint. Disjunctive scheduling constraint signature could be disjunct (e, p) where e represents a tuple of real variables indicating the starting time of each event and $p$ represents a vector of processing times. Then disjunct ( $e, p$ ) could be used to build a concise model for the previous problem, that may be directly used within a modeling language to indicate this particular structure, where $e$ are the variables representing the starting times in which the trucks are going to be attended and p are the processing times for each truck.
disjunct (e, p) can be formulated as the following MILP,

$$
\left.\begin{array}{rl}
e_{j} & \geq e_{i}+p_{i}-M\left(1-b_{i j}\right) \\
e_{i} & \geq e_{j}+p_{j}-M b_{i j}
\end{array}\right\} \forall i, j \text { with } i<j \in[0 . . n]
$$

$$
\begin{aligned}
& b_{i j} \in\{0,1\}, \forall i, j \\
& L \leq e_{i} \leq U, \forall i \in[0 . . n]
\end{aligned}
$$

where $M$ is a large constant (and has to be computed out of the data of the problem, then the continuous relaxation of that constraint will typically give a very weak bound). Binary variables $b_{i j}$ define whether one constraint or the other will be ensured. The combination of $b_{i j}$ with $M$ makes possible that if $b_{i j}=1$ then the second constraint is satisfied independently of the values of $e_{i}$ and $e_{j}$, and if $b_{i j}=0$ then the same happens to the first constraint. The usage of $M$ and $b_{i j}$ allows the creation of a MILP formulation for modeling disjunction of constraints, but uses $O\left(n^{2}\right)$ number of constraints plus the new binary variables.

The fact of knowing the problem structure allow the use of proper methods for tackling the problem. In this case, a flat MILP representation of the disjunctive scheduling would ignore the presence of this kind of structure. Whereas considering explicitly this form of structure allows the application of specific methods like timetabling [67, 132], edge finding [46, 47, 48] and not-first/not-last rules [152], for filtering the variable domains (reduce the domains of the $e_{i}$ as much as possible) and the generation of proper logic-based Benders cuts [90].

### 4.1.5 Product Loading

We now consider a more complex example, where a company must load products from plants to trucks and each product $l \in S$ has a weight $w_{l}>0$ and a cost $\operatorname{cost}_{l, i}$ to be loaded at plant $i \in P$. For each plant $i \in P$ is defined a loading time $t_{i}^{L}$. All the products are available in all plants. For each truck $k \in F$ is defined

- $c_{k}$ capacity in weight of truck $k$,
- cost $_{k}$ cost of using truck $k$

Trucks will be located in a common parking lot. Therefore moving from the parking lot to the plants will take some time, represented by $d_{i, k}$, which is the time $k$ will be available at plant $i$.

The main goal is to decide how the fleet of trucks is going to be used, ensuring that every product $l$ is loaded exactly on one truck and there is not overlapping of loads, minimizing the truck usage cost + product loading cost.

The following MILP model for the previous problem will contain the variables
$x_{i, l, k} \rightarrow$ boolean variable indicating whether $l \in S$ is loaded on $k \in F$ at $i \in P$,
$y_{i, k} \rightarrow$ boolean variable representing whether $k \in F$ will be used at $i \in P$
$s_{i, k} \rightarrow$ loading time of $k \in F$ at $i \in P$.
Assume that global opening is at time 0 and closing at time $C T$, for every plant, then,

$$
\begin{array}{lr} 
& \\
\text { min } \sum_{i \in P} \sum_{k \in F}\left(y_{i, k} \operatorname{cost}_{k}+\sum_{l \in S} x_{i, l, k} \operatorname{cost}_{l, i}\right) & \\
\text { s.t. } &  \tag{4.1.15}\\
& \sum_{l \in S} w_{l} x_{i, l, k} \leqslant c_{k} \\
\sum_{i \in P} \sum_{k \in F} x_{i, l, k}=1 & l \in P, k \in F \\
x_{i, l, k} \leqslant y_{i, k} & i \in S \\
\sum_{i \in P} y_{i, k} \leq 1 & k \in l \in S, k \in F \\
\end{array}
$$

Constraints (4.1.15) enforce that truck capacities are not exceeded and (4.1.16) ensures that each product is loaded exactly once.

Constraints (4.1.17) ensure that if at least one product is loaded on a truck in a certain plant then the corresponding $y_{i, k}$ will be 1 , otherwise will be 0 . Constraints (4.1.18) enforce that trucks are used just once.

Constraints over starting times $x s$ are built using the binary variables $y_{i, k}$ and (4.1.19), enforcing that no overlapping can exist on the plants when scheduling the trucks for loading.

## Structured formulation

Structures like knapsack, semi-assignment and disjunctive scheduling can be detected in the previous MILP model. Constraints (4.1.15) expose 0-1 knapsack packing constraints. Constraints (4.1.16) can be substituted by a semi-assignment constraint and (4.1.18) by a semi-assignment (LEQ) constraint. Moreover, constraints (4.1.19) can be replaced by disjunctive scheduling constraints.

The new formulation will be
s.t.

A structured formulation, like the previous, allows the usage of specialized solution methods for dealing with each different structure (see $\S 4.1 .2, ~ \S 4.1 .2$ and $\S 4.1 .4$ ). It makes the application of relaxation techniques more viable, for splitting the model and being able to attack each structure separately.

## Another structured formulation

Another transformation could be done to the MILP model, consisting in incorporating the time as another dimension to the model, instead of dealing with it as a variable. If all loading times $\left(t_{i}^{\mathcal{L}}\right)$ are $\mathbb{N}$ (also $\mathbb{Q}$, since it is numerable), then one could transform the model, automatically, in such a way the minimum unit of time is 1 .

How to replace the overlapping constraints? Since $y_{i, k}$ will be transformed in $y_{i, k, t}$, then for each $i$ must be verified that for each pair $\left\langle(k, t),\left(k^{\prime}, t^{\prime}\right)\right\rangle$, where $k, k^{\prime} \in F$ and $t \leq t^{\prime}<t+t_{i}^{L}$ $\left(t \in\left[d_{i, k}, C T\right]\right.$ and $\left.t^{\prime} \in\left[d_{i, k^{\prime}}, C T\right]\right)$, the sum $y_{i, k, t}+y_{i, k^{\prime}, t^{\prime}}$ is less or equal than 1 . These constraints define incompatibilities between trucks and times for each plant, therefore it can be modeled with

$$
\begin{align*}
& \min \sum_{i \in P} \sum_{k \in F}\left(y_{i, k} \operatorname{cost}_{k}+\sum_{l \in S} x_{i, l, k} \operatorname{cost}_{l, i}\right) \\
& \text { ks01Pack } \left.\left\{x_{i, l, k}\right\}_{l \in S},\left\{w_{l}\right\}_{l \in S}, c_{k}\right) \quad i \in P, k \in F  \tag{4.1.20}\\
& \operatorname{semiassign}\left(\left\{x_{i, l, k}\right\}_{[l \in S,(i \in P, k \in F)]}\right)  \tag{4.1.21}\\
& x_{i, l, k} \leqslant y_{i, k} \quad i \in P, l \in S, k \in F \\
& \text { semiassignLEQ } \left.\left\{y_{i, k}\right\}_{[k \in F, i \in P]}\right)  \tag{4.1.22}\\
& \operatorname{disjunct}\left(\left\{s_{i, k}\right\}_{k \in F},\left\{\left\{_{i}^{L} y_{i, k}\right\}_{k \in F}\right) \quad i \in P\right.  \tag{4.1.23}\\
& s_{i, k} \in\left[d_{i, k}, C T\right] \subset \mathbb{R} \\
& x_{i, l, k} \in\{0,1\} \\
& y_{i, k} \in\{0,1\}
\end{align*}
$$

$$
\begin{aligned}
& \left.\begin{array}{rl}
s_{i, k} & \geq s_{i, k^{\prime}}+t_{i}^{\mathcal{L}} y_{i, k^{\prime}}-M\left(1-b_{i, k, k^{\prime}}\right) \\
s_{i, k^{\prime}} & \geq s_{i, k}+t_{i}^{\mathcal{L}} y_{i, k}-M b_{i, k, k^{\prime}}
\end{array}\right\} \quad \begin{array}{r}
i \in P, \\
\left(k, k^{\prime}\right) \in F^{2} \mid k^{\prime}<k
\end{array} \\
& s_{i, k} \in\left[d_{i, k}, C T\right] \subset \mathbb{R} \\
& x_{i, l, k} \in\{0,1\} \\
& y_{i, k} \in\{0,1\} \\
& b_{i, k, k^{\prime}} \in\{0,1\}
\end{aligned}
$$

indset ( x , cond) where x could be the variables $y_{i, k, t}$ and cond the logical condition $t \leq t^{\prime}<t+t_{i}^{L}$ which defines the pairs of variables that are incompatible.

Dimension $t$ in all constraint must be $\geq d_{i, k}$, to enforce that no truck is used before its arrival time to the plant.

Then the formulation should be

$$
\min \sum_{i \in P} \sum_{k \in F} \sum_{t \in\left[d_{i, k}, C T\right]}\left(y_{i, k, t} \operatorname{cost}_{k}+\sum_{l \in S} x_{i, l, k, t} \operatorname{cost}_{l, i}\right)
$$

$$
\begin{align*}
& \text { s.t. } \\
& \operatorname{ks01Pack}\left(\left\{x_{i, l, k, t}\right\}_{l \in S_{i}},\left\{w_{l}\right\}_{l \in S_{i}}, c_{k}\right) \quad i \in P, k \in F, t \in\left[d_{i, k}, C T\right] \\
& \operatorname{semiassign}\left(\left\{x_{i, l, k, t}\right\}_{\left[l \in S,\left(i \in P, k \in F, t \in\left[d_{i, k}, C T\right]\right)\right]}\right) \\
& \begin{array}{r}
i \in P, l \in S, k \in F, \\
x_{i, l, k, t} \leqslant y_{i, k, t} \\
t \in\left[d_{i, k}, C T\right]
\end{array} \\
& \text { semiassignLEQ }\left(\left\{y_{i, k, t}\right\}_{\left[k \in F,\left(i \in P, t \in\left[d_{i, k}, C T\right]\right)\right]}\right)  \tag{4.1.24}\\
& \text { indset }\left(\left\{y_{i, k, t}\right\}_{k \in F, t \in\left[d_{i, k}, C T\right]}\right. \text {, } \\
& \left.t \leq t^{\prime}<t+t_{i}^{L}\right)  \tag{4.1.25}\\
& x_{i, l, k, t} \in\{0,1\} \\
& y_{i, k, t} \in\{0,1\}
\end{align*}
$$

In this case, structure in 4.1.23 is substituted by several incompatibility constraints 4.1.25, that avoid the overlapping.

Let be $n=\max \left\{|P|,|S|,|F|,\left|\left[d_{i, k}, C T\right]\right|\right\}$, then one can estimate the amount of variables as $O\left(n^{3}\right)$ and the amount of constraints as $O\left(n^{3}\right)$ in formulation 4.1.5, whereas in this formulation (4.1.5) the amount of variables and constraints are increased to $O\left(n^{4}\right)$, due to the addition of time as a dimension.

Despite the increase of constraints and variables, in some cases it might be more viable to solve the problem through formulation 4.1.5; mainly when the amount of time intervals is no so big, since in 4.1.5 scheduling constraints are substituted by independent set constraints. Which, due to the elimination of "big M" on the MILP formulation, may provide better continuous bounds on a LP relaxation.

## Yet another structural formulation

Instead of using structures (4.1.24) and (4.1.25), one could use only one.
A semi-assignment constraint, when it does not enforce equality, can be seen as a set of incompatibility constraints that defines a complete graph. Then adding the proper condition in (4.1.25) the resulting incompatibility graph will containt cliques representing the semi-assignment constraints. Then the resulting model would be

$$
\min \sum_{i \in P} \sum_{k \in F} \sum_{t \in\left[d_{i, k}, C T\right]}\left(y_{i, k, t} \operatorname{cost}_{k}+\sum_{l \in S} x_{i, l, k, t} \operatorname{cost}_{l, i}\right)
$$

s.t.

$$
\begin{array}{lr}
\operatorname{ks01Pack}\left(\left\{x_{i, l, k, t}\right\}_{l \in S_{i}},\left\{w_{l}\right\}_{l \in S_{i}}, c_{k}\right) & i \in P, k \in F, t \in\left[d_{i, k}, C T\right] \\
\operatorname{semiassign}\left(\left\{x_{i, l, k, t}\right\}_{\left[l \in S,\left(i \in P, k \in F, t \in\left[d_{i, k}, C T\right]\right)\right]}\right) & \\
x_{i, l, k, t} \leqslant y_{i, k, t} & \\
\operatorname{indset}\left(\left\{y_{i, k, t}\right\}_{k \in F, i \in P, t \in\left[d_{i, k}, C T\right]},\right. & \\
\quad\left(t \leq t^{\prime}<t+t_{i}^{L} \wedge i=i^{\prime}\right) \vee &  \tag{4.1.26}\\
\left.\quad\left(k=k^{\prime} \wedge i \neq i^{\prime} \wedge t \neq t^{\prime}\right)\right) &
\end{array}
$$

$$
\begin{aligned}
& x_{i, l, k, t} \in\{0,1\} \\
& y_{i, k, t} \in\{0,1\}
\end{aligned}
$$

(4.1.26) is a single constraint that embraces constraints (4.1.24) and (4.1.25).

This formulation uses less structures, concentrating two former structures on one huge structured constraint. However this formulation relies on the efficiency of the solvers for tackling this huge constraint.

### 4.2 Shared Variables - Blocks - Relaxation

A block describes a tree where each substructure node must be a $\operatorname{LfP}\left(\right.$ or $\operatorname{LfP}{ }^{\mathcal{L}}$ ) or a block. A block also enables the controlled sharing of variables. This sharing is managed by the template patterns (see $\S 1.2 .3$ ), which define how the variables exported by the substructures must be arranged.

Several cases of template patterns may appear, Let $X_{i}, Y_{j}$ be template items then the following links may be built,

1. $\left[X, Y_{1}\right], \ldots,\left[X, Y_{n}\right]$,
2. $\left[X_{1}\right], \ldots,\left[X_{n}\right]$ and $\left[X_{1}, \ldots, X_{n}\right]$,
3. $\left[X_{1}, X_{2}\right],\left[X_{2}, X_{3}\right], \ldots,\left[X_{n}, X_{n+1}\right]$

For dealing with shared variables, relaxations could be applied. When the amount of shared variables is large and they are partitioned over all substructures except for one that contain all shared variables (these substructure can be also called linking substructure), then Lagrangian Relaxation [111] can be applied over these linking substructures considering them in the cost function, and allowing to deal with the independent substructures separately. When in presence of multicommodity network flow problems, a specialized Interior-Point method [51, 98] can be applied. In [83] there is an application of parallel Interior-Point for dealing with block-structured matrices.

When the formulation is linear a graphic representation of shared variables spread over linking substructures, can be seen in figure 4.2 a ).


Figure 4.2: a) Shared variables partitioned over substructures (except for one) in a linear model b) Variables shared by all substructures in a linear model

If shared variables are just a few and distributed over many substructures, see figure 4.2 b ), something like Benders Decomposition [33] (extended for nonlinear [77]) may be applied.

### 4.2.1 Returning to the Product Loading problem

Returning to the formulation made in $\S 4.1 .5$, one can make a diagram relating constraints with variables (see Figure 4.3).


Figure 4.3: Graphic relating constraints with variables for formulation §4.1.5
In figure 4.3 the following structures are represented
A 0-1 knapsack packing constraints and semi-assignment constraints
B unstructured inequalities
C semi-assignment constraints
D disjunctive scheduling constraints
One can apply Lagrangian Relaxation to eliminate linking constraints B, separating A from D and C.

Since B represents the constraints $x_{i, l, k} \leq y_{i, k}, \forall i \in P, l \in S, k \in F$, then when spanned will look like,

$$
\left(\begin{array}{ccccccccccc} 
 \tag{4.2.1}\\
& & & & & & & & & & \\
\\
& & & & & \vdots & & & & &
\end{array}\right) \quad\left(\begin{array}{c}
\vdots \\
x_{i, l, k} \\
\vdots \\
y_{i, k} \\
\vdots
\end{array}\right) \leqslant\left(\begin{array}{c}
\vdots \\
0 \\
\vdots \\
0 \\
\vdots
\end{array}\right)
$$

The number of equations in 4.2 .1 is $|P| *|S| *|F|$. If one multipies (4.2.1) by the Lagrangian multipliers $\left(\cdots \lambda_{i, l, k} \cdots\right)$ then one can obtain the following,

$$
\begin{equation*}
\sum_{i \in P} \sum_{l \in S} \sum_{k \in F}\left[x_{i, l, k} \lambda_{i, l, k}\right]-\sum_{i \in P} \sum_{l \in S} \sum_{k \in F}\left[y_{i, k} \lambda_{i, l, k}\right] \tag{4.2.2}
\end{equation*}
$$

Then adding (4.2.2) to the objective function and eliminating constraints in B , the problem will be separated in the following two sub-problems,

$$
\begin{align*}
& \quad \min \sum_{i \in P} \sum_{l \in S} \sum_{k \in F}\left[x_{i, l, k}\left(\operatorname{cost}_{l, i}+\lambda_{i, l, k}\right)\right] \\
& \text { s.t. } \\
& \quad \text { ks01Pack }\left(\left\{x_{i, l, k}\right\}_{l \in S_{i}},\left\{w_{l}\right\}_{l \in S_{i}}, c_{k}\right)  \tag{4.2.3}\\
& \quad \text { semiassign }\left(\left\{x_{i, l, k}\right\}_{[l \in S,(i \in P, k \in F)]}\right) \\
& \quad x_{i, l, k} \in\{0,1\}
\end{align*}
$$

$$
\begin{array}{ll}
\quad \min \sum_{i \in P} \sum_{k \in F}\left[y_{i, k}\left(\operatorname{cost}_{k}-\sum_{l \in S} \lambda_{i, l, k}\right)\right] & \\
\text { s.t. } & \\
\quad \text { semiassignLEQ }\left(\left\{y_{i, k}\right\}_{[k \in F, i \in P]}\right) &  \tag{4.2.4}\\
\quad \operatorname{disjunct}\left(\left\{s_{i, k}\right\}_{k \in F},\left\{t_{i}^{L} y_{i, k}\right\}_{k \in F}\right) & \\
\quad s_{i, k} \in\left[d_{i, k}, C T\right] \subset \mathbb{R} & \\
& y_{i, k} \in\{0,1\}
\end{array}
$$

For solving subproblem (4.2.4) one can apply Benders Decomposition [33, 77]. The master problem can be written,

$$
\begin{equation*}
\min \sum_{i \in P} \sum_{k \in F}\left[y_{i, k}\left(\operatorname{cost}_{k}-\sum_{l \in S} \lambda_{i, l, k}\right)\right] \tag{4.2.5}
\end{equation*}
$$

s.t.
semiassignLEQ $\left(\left\{y_{i, k}\right\}_{[k \in F, i \in P]}\right)$
Benders Cuts
$y_{i, k} \in\{0,1\}$
If $y_{i, k}^{h}$ is the solution of (4.2.5) at iteration h , then the subproblem separates into the following scheduling problem for each plant.

$$
\begin{align*}
& \quad \min \sum_{i \in P} \sum_{k \in F}\left[y_{i, k}^{h}\left(\operatorname{cost}_{k}-\sum_{l \in S} \lambda_{i, l, k}\right)\right] \\
& \text { s.t. }  \tag{4.2.6}\\
& \quad \text { disjunct }\left(\left\{s_{i, k}\right\}_{k \in F},\left\{t_{i}^{L} y_{i, k}^{h}\right\}_{k \in F}\right) \quad i \in P \\
& \quad s_{i, k} \in\left[d_{i, k}, C T\right] \subset \mathbb{R}
\end{align*}
$$

Note that the objective function in (4.2.6) is a constant, then one will be looking for a valid schedule for each plant. If no feasible solution is found for plant $i$ then the following cut will be added to the master,

$$
\sum_{k \in K_{i, h}}\left(1-y_{i, k}\right) \geq 1
$$

where $K_{i, h}=\left\{k \in F \mid y_{i, k}^{h}=1\right\}$ are the trucks scheduled in $i \in P$. These cuts will exclude at least one truck in $K_{i, h}$.

To solve (4.2.3) one can also apply Benders Decomposition, taking as master the semi-assignment problem. In this case solving the subproblem (knapsack) will be just verifying that the solution of the master is feasible for the knapsack constrains, because they share the same variables.

Then solutions of subproblems (4.2.3) and (4.2.4) will be used by the Lagrangian dual for obtaining the solution of the problem.

### 4.2.2 Tackling the other formulations

For formulations made in $\S 4.1 .5$ and $\S 4.1 .5$, one can also make diagrams relating constraints with variables, Figure 4.4.

In figure 4.4 a ) are represented the following structures
A 0-1 knapsack packing constraints


Figure 4.4: Graphic relating constraints with variables for formulation a) $\S 4.1 .5$ and b) $\S 4.1 .5$

B unstructured inequalities
C semi-assignment constraints
D independent set constraints
in this case if one relaxes constraints in B then one will obtain a Lagrangian term similar to (4.2.2), only that the amount of Lagrangian multipliers will be $O\left(n^{4}\right)$ instead of $O\left(n^{3}\right)$. The model then will be divided in two subproblems the first similar to (4.2.3) and the second will be something like,

$$
\begin{array}{ll}
\quad \min \sum_{i \in P} \sum_{k \in F} \sum_{t \in\left[d_{i, k}, C T\right]}\left[y_{i, k, t}\left(\operatorname{cost}_{k}-\sum_{l \in S} \lambda_{i, l, k, t}\right)\right] & \\
\text { s.t. } & \\
\quad \text { semiassignLEQ }\left(\left\{y_{i, k, t}\right\}_{\left[k \in F,\left(i \in P, t \in\left[d_{i, k}, C T\right]\right)\right]}\right) &  \tag{4.2.7}\\
\quad \operatorname{indset}\left(\left\{y_{i, k, t}\right\}_{k \in F, t \in\left[d_{i, k}, C T\right]},\right. & i \in P \\
\left.\quad t \leq t^{\prime}<t+t_{i}^{L}\right) &
\end{array}
$$

Subproblem (4.2.7) can be solved applying Benders Decomposition, but a branch and relax scheme, solving the continuous relaxation at each step, could also be applied. If the solution of the continuous relaxation is feasible in (4.2.7), then it is the optimal; otherwise cuts are generated analyzing which constraints are violated. As a consequence, the resulting continuous relaxation might give a tighter bound.

In figure 4.4 b ) are represented the following structures
A 0-1 knapsack packing constraints
B unstructured inequalities
C independent set constraints
when eliminating B, constraints A remain the same, but in C there will be only one complicated constraint, which, nevertheless is structured and can be tackled using algorithms for independent set constraints §4.1.2.

### 4.3 Solvers' Tree

The previous section showed how solution methods applied to structured models are far from simple and linear concatenations of solvers that deal with the sub-problems. Since the block structure
gives the model a tree fashion, the solution methods must be also assigned and executed using a similar fashion.

Consider we have a d_B_Lagrangian_Relax_C block (§1.3), that ensures a variable and substructure disposition suitable for applying Lagrangian Decomposition. Assume we have a solver to deal with this kind of block. Note that when arrived to the point of solving the sub-problems, the solver may delegate on sub-solvers assigned to the corresponding substructures. Then we also need to assign a solver to the substructures, except when in presence of the modifier d_loc (see §1.2.3), that explicitly says that the modified substructure will be tackled directly by the block's solver.

Returning to the d_B_Lagrangian_Relax_C block, note that solvers must be assigned to the first substructure, besides the one assigned to the block itself. But if we assume the the first substructure is also a block, this will generate a solver assignment that will look like a tree (see Figure 4.5).


Figure 4.5: Solver Tree assignment
Besides the structured instance, I-DARE(solve) (the solving part of I-DARE), will also take as input the way solvers are assigned to the structures. The Solver's Tree specifies how solvers must be later called, for obtaining the problem's solution. In the following definition we will use the concept of solver registered to a structure class, that will be formalized in the next section.

Definition 4.3.1 (Solvers' Tree of a Leaf Problem) Let ?P:?C, ?C::d_LeafProblem_C and I a list of already fixed indices, then we say that the Solvers' Tree of ?P is defined by the term _st ( $s_{1}, \ldots, s_{k}$ ), where $\mathbf{s}_{i}$ is a solver registered to ?C, $i \in[1 . . k]$, and $k=\|$ ?P.freeinds $\|-\| I \|$.

Definition 4.3.2 (Solvers' Tree of a Block) Let ?B:?C, ?C:: d_Block_C and I a list of already fixed indices, then we say that the Solvers' Tree of ?B is defined by the term _st $\left(\mathrm{s}_{1}\left(\mathrm{st}_{0}^{1}, \ldots, \mathrm{st}_{m}^{1}\right), \ldots, \mathrm{s}_{k}\left(\mathrm{st}_{0}^{k}, \ldots, \mathrm{st}_{m}^{k}\right)\right)$, where $\mathrm{s}_{i}$ is a solver registered to $? \mathrm{C}, i \in[1 . . k], k=\| ?$ P.freeinds $\| ;$ and $\mathrm{st}_{j}^{i}$ is the Solvers' Tree of ?B.subs $[j]$ considering $I \cup$ ?B.freeinds as the list of already fixed indices, $j \in[0 . . m], m=\mid$ ?B.subs $\mid-1$.

Definition 4.3.3 (Solvers' Tree of a Formulation) Let ?F::d_Formulation and ?C the class of ?F.root (?F.root:?C), then we say that the Solvers' Tree of ?F if equal to the Solver's Tree of ?F.root considering $\emptyset$ as the list of already fixed indices.

For instance consider the formulation diagram in Figure 4.6, then if we assume $\|\|\|=2\| 1\|=3$,


Figure 4.6: Formulation Diagram
and $\|12\|=2$, then the following Solvers' Tree may be generated,

```
-st(solver(A)1(
    -st(solver(B) 1,1(
        -st(solver(D) 1,1),
            -st(solver(E) 1,1,1
                solver(E) 1,1,2
            solver(B)_1,2(
            -st(solver(D) 1,1).
            -st(solver(E) 1,2,1
                solver(E) 1,2,2
            solver(B) (1,3(
            -st(solver(D) 1,1),
            -st(solver(E) 1,3,1
                solver(E) (1,3,2
                    )
    -st(solver(C) ()
    solver(A)2(
    -st(solver(B) 2,1(
            -st(solver(D) 2,1).
            -st(solver(E) 2,1,1
                solver(E)2,1,2
            solver(B) 2,2(
            -st(solver(D) 2,1),
            -st(solver(D) 2,1),
            -st(solver(E)
                solver(E)
            solver(B)2,3(
            -st(solver(D)2,1),
            -st(solver(D) 2,1),
                solver(E) 2,3,2)
                )
    -st(solver(C)2)
)
```


### 4.4 Solvers and Structures

As could be seen in the previous sections, explicitly recognizing the structures in Mathematical Models, allows us to apply specific and efficient solution methods. This is one of the reasons behind the explicit structure manipulation I-DARE has. The I-DARE(solve) package, defines an extensive library of solution methods, that will be related to the structure classes defined in I-DARE(lib). I-DARE(solve) will be composed of a $\mathrm{C}++$ side and a $\mathcal{F}$ LORA-2 side. A solver must have a $\mathrm{C}++$ wrapper that integrate it to the system, and at the same time must have a $\mathcal{F}$ LORA- 2 representation.

Figure 4.7 represents how solvers are organized in $\mathrm{C}++$ and $\mathcal{F}$ LORA-2. The main idea in the $\mathrm{C}++$ part is to use a dynamic library loading mechanism to add new solution methods to the system. We have two kinds of registrations, solver registration and blueprint registration.

The solver registration adds a new solution method to a specific structure class. This solution method may then be used to form the Solvers' Tree and finally solve the problem. On the other hand, the blueprint registration defines the representative solver for a structure class. The blueprint is simply a solver that inherits directly from c_solver, it defines the beginning of the sub-hierarchy of solvers for tackling the structure class the blueprint is registered to. A structure class is represented by only one blueprint. If by any mistake there are two blueprints representing the same structure class, I-DARE(solve) will report a warning and will ignore one of them (so it will ignore all the sub-hierarchy that descends from that blueprint).

Observe, each solver makes reference to an object of type c_solution. This class will represent a basic solution containing only a dictionary of the variables and their values. It is well known that some solvers may provide more information as a solution, for instance the dual values of the constraints. One may create, a new class of solutions inheriting from c_solution, that includes those dual values. Hence, there will also be a solution hierarchy.

Each solver must export themselves to the $\mathcal{F}$ LORA-2 solvers' hierarchy, specifying an object instance of the following class,

Listing 4.1: Solver Root Class


Figure 4.7: C++ vs. FLORA-2 hierarchies

```
[
    structure }\quad> d_Component_C
    confTemplate => _term
].
```

where,

- structure - is the ?C::d_Component_C class to which the solver will be registered to,
- confTemplate - defines how the configuration of the solver may be (see 4.4.2).

But if the solver represents a blueprint, it is responsible of exporting a little more than that. Each blueprint must also export an object instance of the following class,

Listing 4.2: Blueprint exported to the $\mathcal{F}$ LORA-2 file

```
d_blueprint
[
    structure => d_Component_C,
    retType => d_solution ,
    ?signature1,
    ?signatureK
].
```

where

- structure is the representative structure class,
- retType is d_solution or any ?X such that ?X:: d_solution,
- ? signaturel is a property with the following format ?name(?argtypes) -> ?retType where
- ?name is a method identifier included by the blueprint (in I-DARE(solve)),
- ?argtypes is an ordered comma list of the argument type names of each parameter of ?name,
- ?retType is the return type name of ?name.

When a blueprint does not include any new method (with respect to c_solver) then there will be no signatures.

I-DARE(solve) offers a basic class to create a solution, d_solution. However, a solver may extend this class to return more richer solutions (for example, including dual values). This solution hierarchy must also be exported automatically to $\mathcal{F}$ LORA-2. For instance if we have, the I-DARE(solve) class solution_dual that extends from d_solution, then the following will be generated

Listing 4.3: Example of solution hierarchy
1 solution_dual : d_solution.

### 4.4.1 Solvers exported and new Structure relations

The $\mathcal{F}$ LORA-2-side of I-DARE(solve), has two main purposes. The first one is to provide the bases for the Solvers' Tree generation, that will mainly use the d_solver objects. The second is to define a new type of relations between structure classes.

Let's go back to the d_B_Lagrangian_Relax_C class,

```
d_B_Lagrangian_Relax_C :: d_Block_C [
    ids }->\mathrm{ [ sub, linking],
    subsC -> [ d_LR_C, d_loc(d_Linear_Constraints_C)],
    link > [([X],d_all), ([X],d_all)]
].
d_LR_C :: d_Component_C [abstract]. // Auxiliary structure class
```

With the inheritance relation (:: ) used so far, the only way of creating an instance of d_B_Lagrangian_Relax_C is by creating instances of d_LR_C. But that would be impossible, since d_LR_C is abstract. Therefore, I-DARE defines a new way of dynamically verifying for structure classes equivalence based on the blueprint specification.

Besides all semantical meaning a structure class may have, the most important characteristic is the way its blueprint is defined (which solution type it provides, which methods it defines). For instance,

```
d_LR_C_blueprint : d_blueprint
[
    structure -> d_LR_C,
    retType -> solution_dual,
    //signatures
    add_to_OF(['vector <double >']) -> 'void'
].
```

defines the blueprint for d_LR_C. Notice the signature add_to_OF(['vector <double>']) -> 'void', that specifies that all solvers descending from that blueprint must implement/use that method. This method is intended to update the objective function of the structure being solved, functionality important in a Lagrangian Relaxation solution method.

Now, if another structure class has a blueprint that contains at least that signature and any descendant of that solution type, the system will consider it of type d_LR_C. To formalize this concept we will use the blueprint equivalence relation. But first we need to define when to signatures are equivalent. For doing so, we will follow a naive approach, meaning that if two signatures are identical the system will assume that the corresponding methods will perform equivalent actions. There will not be sub-typing rules [133] defined for argument types and return type (this will be left for future research).

Therefore two signatures ?n1(?at1)->?rt1 and ?n2(?at2)->?rt2 will be considered equivalent iff ? $\mathrm{n} 1=$ ? n 2 , ? at1=?at2 and ? $\mathrm{rt} 1=$ ? $\mathrm{rt2}$.

Definition 4.4.1 (Blueprint equivalence relation rule) Given two blueprints $B_{1}$ and $B_{2}, B_{2}$ is equivalent to $B_{1}$, denoted by $B_{2} \leadsto B_{1}$ iff

- $B_{2}$.retType :: $B_{1}$.retType and
- $\forall\left(? n(? a t)->? r t \in B_{1}\right) \exists\left(? \mathrm{n}^{\prime}\left(? a t^{\prime}\right)->? r t^{\prime} \in B_{2}\right)$ such that $? \mathrm{n}(? a \mathrm{at})->? \mathrm{rt}$ and $? \mathrm{n}$ '(?at')->?rt' are equivalent.

For instance,

```
d_MCF_C_blueprint : d_blueprint
[
    structure -> d_MCF_C,
    retType -> solution_dual,
    //signatures
    add_to_OF(['vector<double>']) -> 'void',
].
```

may be the blueprint of d_MCF_C class, and it is equivalent to d_LR_C_blueprint, due to rule 4.4.1.
Proposition 4.4.2 ( $\sim$ - property) Let $\mathcal{B}$ be the set of all blueprints, then the $\leadsto$ relation over $\mathcal{B} \times \mathcal{B}$ is a partial order relation over $\mathcal{B}$.

Proof. For $\sim$ to be a partial order it must be reflexive, antisymmetric and transitive.

- $\forall(B \in \mathcal{B}) B$.retType :: $B$.retType and $\forall$ ? $\mathrm{n}($ ?at $)->$ ? $\mathrm{rt} \in B$, it is equivalent to itself, then $B \sim B$. Therefore $\leadsto$ is reflexive.
- $\forall\left(B_{1}, B_{2} \in \mathcal{B}\right)$ if $B_{1} \leadsto B_{2}$ and $B_{2} \leadsto B_{1}$ then
- $B_{1}$.retType :: $B_{2}$.retType and $B_{2}$.retType :: $B_{1}$.retType, thus $B_{1}$.retType $=B_{2}$.retType, due to :: definition;
 equivalent, and $\forall\left(? \mathrm{n}(? \mathrm{at})->\right.$ ? $\left.\mathrm{rt} \in B_{2}\right) \exists\left(\right.$ ? $\mathrm{n}^{\prime}\left(? a t^{\prime}\right)->$ ?rt' $\left.\in B_{1}\right)$ such that $? \mathrm{n}(? \mathrm{at})->$ ? rt and ?n'(?at')->? ?rt' are equivalent, thus $B_{1}$ 's signature set is equal to $B_{2}$ 's signature set, due to set definition and signature equivalence definition.

Hence, $B_{1}=B_{2}$. Therefore $\sim \sim$ is antisymmetric.

- $\forall\left(B_{1}, B_{2}, B_{3} \in \mathcal{B}\right)$ if $B_{3} \leadsto B_{2}$ and $B_{2} \leadsto B_{1}$ then
- $B_{3}$.retType :: $B_{2}$.retType $:: B_{1}$.retType, thus $B_{3}$.retType :: $B_{1}$.retType, due to :: definition;
$-\forall\left(? \mathrm{n}(? a \mathrm{at})->? \mathrm{rt} \in B_{1}\right) \exists\left(? \mathrm{n}^{\prime}\left(? a t^{\prime}\right)->? \mathrm{rt} \mathrm{t}^{\prime} \in B_{2}\right)$ such that $? \mathrm{n}(? a \mathrm{at})->? \mathrm{rt}$ and $? \mathrm{n}^{\prime}\left(? a t^{\prime}\right)->? \mathrm{rt}{ }^{\prime}$ are
 and ?n'"(?at')->? $\mathrm{rt}^{\prime \prime}$ are equivalent, thus $\forall\left(? \mathrm{n}(? \mathrm{at})->? \mathrm{rt} \in B_{1}\right) \exists\left(? \mathrm{n}\right.$ "'(? at'")->?rt" $\left.\in B_{3}\right)$ such that ?n(?at)->?rt and ?n"'(? at'")->?rt" are equivalent, passing through the ?n'(?at')->?rt' $\in$ $B_{2}$ ).

Hence, $B_{3} \leadsto B_{1}$. Therefore $\leadsto$ is transitive.

The fact $\leadsto$ is a partial order as well as ::, suggests us that a simple extension of :: can be done, to ensure that if two blueprints are equivalent, it means that the corresponding structure classes are equivalent as well. The point being that when a structure ?P is present inside a block, the only thing that block cares about is that its corresponding solvers are able to use the solver of ?P. So, the block is only interested in whether the two structure classes have equivalent blueprints.

To make this process transparent to the user, I-DARE uses this dynamic extension of the $\mathcal{F}$ LORA-2 operator ::.

Listing 4.4: Extension of :: $\mathcal{F}$ LORA-2 operator

```
?X::?C :- ?X::d_Component_C[not abstract],
    ?C::d_Component_C[abstract],
    ?xb : d_blueprint[structure -> ?X],
    ?cb : d_blueprint[structure }->\mathrm{ ? ?C],
    ? xb ~ ?cb.
```

So now the fact of d_LR_C being abstract, will not longer be a problem, since d_MCF_C :: d_LR_C will be true due to Listing 4.4. Therefore all objects ?O:d_MCF_C will automatically be considered ?O:d_LR_C, so one may use ?O inside a d_Lagrangian_Relax_C block.

Proposition 4.4.3 Let be ? $C_{1}:: d_{-}$Component_C[abstract] and ? $C_{2}:: d$ d_Component_C[not abstract], such that ?cb. : d_blueprint [structure $->? C_{1}$ ] and ?cb $b_{2}$ : d_blueprint [structure $->? C_{2}$ ], then the usage of a ?O:?C $C_{2}$ in a place where ? $C_{1}$ is required do not create any well-formedness problem iff ?cb $b_{2} \leadsto ? c b_{1}$.

Proof. To prove this proposition, let's consider the places where this substitution may create any problem, and there is only one place, wherein ? $\mathrm{C}_{1}$ is a substructure of a certain block, so ?O:? $\mathrm{C}_{1}$ must be true to ensure well-formedness. And in fact it is true, since ?O:? $\mathrm{C}_{2}$ and ? $\mathrm{cb}_{2} \sim ? \mathrm{cb}_{1}$, so ? $\mathrm{C}_{2}:$ ? $\mathrm{C}_{1}$ (due to Listing 4.4). Therefore, ? $\mathrm{O}: ? \mathrm{C}_{1}$ is true.

Note also that to consider the "blueprint equivalence" generated class relation do no create any semantic problem (due to the substitution of one structure by another), because the structure we are replacing will always be abstract. Abstract structures do not represent any "directly solvable" class of problems, generally they refer to a wide set of problems defined by how the provide their solution and the way other classes can interact with them.

### 4.4.2 Configuration Templates

As previously mentioned, each solver must define how it must be configured. For this purpose I-DARE(solve) defines Configuration Templates (CT). A CT is a hierarchical structure defining the relevant algorithmic parameters and the possible range of their values. Hence, CTs can be easily used to describe single configurations by simply forcing each parameter to have a singlevalued domain.

Two descriptions of CTs are available: the "external" and the "internal" one. The external one is in terms of an XML file that specifies parameters and their domains. CTs currently support four base parameter types: integer, double, choice and vector. The integer type defines bounds of the parameter and a default value. For instance,

Listing 4.5: Example of Integer parameter type
1
<INT name $=$ "param1" bounds $=" 0: 1,4,8: 10 "$ def_val $=" 0 " />$
defines an integer parameter that can take values $0,1,4$ and from 8 to 10 , with default value set to 0 . The attribute bounds will be a list composed by integer numbers or pairs of the form I:u. Each pair I:u specifies a lower and upper bound of a subset in the domain of the parameter. The elements of the bounds list must be disjoint.

The double type defines also bounds and default value. It also includes a step that specifies the increment that will be used to iterate between the bounds. For instance,

Listing 4.6: Example of Double parameter type
$<$ DOUBLE name $="$ param2" bounds $=" 0: 0.01: 1,4: 6.5 "$ def_val $=" 0 " />$
defines a double parameter that can take values between 0 and 1 , with increment (step) set to 0.01 and between 4 and 6.5 also with step 0.01 , and default value set to 0 . Note that in this case there is a new kind of element in the bounds list, $\mathrm{I}: \mathrm{s}: \mathrm{u}$, that represents a lower bound, step and upper bound. When there is an interval without step, in the bounds list, the minimum step (present in the list) will be taken; but if there is no step defined at all, a system predefined step will be assumed.

The vector type defines a parameter that may contain a multi-dimensional vector. It specifies the dimension bounds, the type of each vector's element and a default value. For instance,

Listing 4.7: Example of Integer parameter type

```
<VECTOR name=" param3" dims = " 2| 3">
    <INT name="internal1" bounds="-1:100" def_val = "3"/>
    <DEF_VAL dims=" 2| ">
    <V val = "-1"/>
    <V val = "2"/>
    <V val = "34"/>
    <V val = "4"/>
    </DEF_VAL>
</VECTOR>
```

defines a vector parameter with two dimensions, the first one bounded to 2 and the second one to 3. It also specifies that each vector's element must be of integer type (with the stated bounds and default value). Note that a vector type's default value is defined using a vector tag that sets the dimensions (respecting the bounds) and each element. The elements are represented in a linear form, even if it has more than one dimension; of course, the total amount of elements must be equal to $\Pi_{d \in \text { dimension }}\|d\|$.

Finally, the choice type defines a nominal parameter that may be used to describe, for instance, a method to be used inside the solver. This type specifies all the nominal values it may take. Each nominal value is a configuration template as well. For instance,

Listing 4.8: Example of Choice parameter type

```
<CHOICE name=" param4" def_val = "choice1">
        \(<\mathrm{E}\) name \(=\) "choice1"/>
    \(<\) E name \(=\) "choice2" \(>\)
            <DOUBLE name="subparam1" bounds=" 0.01:0.005:0.5" def_val = "0.05"/>
    </E>
    \(</\) CHOICE \(>\)
```

defines a domain of two choices. The second choice specifies a sub-parameter of type double; within the tag $E$, a whole CT may appear. This allow us to create hierarchical configurations, where the set of parameters in the configuration is not always the same (although of course the total set of parameters which may appear in any configuration pertaining to a CT is fixed). This is useful because solvers may support more than one different algorithm, and some parameters may not have a meaning for some of them. For instance, Linear Program solvers may employ either simplex approaches or interior-point ones; while some algorithmic parameters are typically common to both approaches, there are others that only make sense for one of them.

When a solver is exported to the $\mathcal{F}$ LORA-2 file, it exports also its CT. Therefore, the "internal" representation of the CT in $\mathcal{F}$ LORA-2 is automatically constructed as an instance of the classes: configuration_T, conflnt_T, confDouble_T, confVector_T and confChoice_T,

```
configuration_T.
confNumber_T [
    bounds => _list,
    def_val => _number
].
confInt_T :: confNumber_T.
confDouble_T :: confNumber_T.
```

```
confVector_T [
    dims => _list,
    type => confNumber_T,
    def_val => [_list, _list]
] .
confChoice_T [
    def_val => configuration_T
].
```

For instance, the $\mathcal{F}$ LORA-2 file corresponding to the parameter types exposed in Listings 4.5, 4.6, 4.7 and 4.8 would look like

```
param1 : conflnt_T [
    bounds
-> [(0,1), 4, (8,10)],
    def_val -> 0
].
param2: confDouble_T [
    bounds
->[(0,0.01,1), (4,6.5)]
    def_val -> 0
].
internal1: conflnt_T [
    bounds }->> [(-1,100)]
    def_val -> 3
].
param3: confVector_T [
    type -> internal1,
    dims }->\mathrm{ [2 ,3],
    def_val -> [[2,2],[-1,2,34,4]]
].
```

```
choice1 : configuration_T.
```

choice1 : configuration_T.

```
choice1 : configuration_T.
subparam1 : confDouble_T [
subparam1 : confDouble_T [
subparam1 : confDouble_T [
    bounds
    bounds
    bounds
\(\rightarrow\) [(0.01, 0.005, 0.5)],
\(\rightarrow\) [(0.01, 0.005, 0.5)],
\(\rightarrow\) [(0.01, 0.005, 0.5)],
    def_val \(\rightarrow 0.05\)
    def_val \(\rightarrow 0.05\)
    def_val \(\rightarrow 0.05\)
].
].
].
subparam1 : choice2.
subparam1 : choice2.
subparam1 : choice2.
choice2 : configuration-T.
choice2 : configuration-T.
choice2 : configuration-T.
choice1 : param4.
choice1 : param4.
choice1 : param4.
choice2 : param4.
choice2 : param4.
choice2 : param4.
param4 : confChoice_T [
param4 : confChoice_T [
param4 : confChoice_T [
    def_val \(->\) choice1
    def_val \(->\) choice1
    def_val \(->\) choice1
]
]
]
param1 : confTemplate1.
param1 : confTemplate1.
param1 : confTemplate1.
param2 : confTemplate1.
param2 : confTemplate1.
param2 : confTemplate1.
param3 : confTemplate1.
param3 : confTemplate1.
param3 : confTemplate1.
param4 : confTemplate1.
param4 : confTemplate1.
param4 : confTemplate1.
confTemplate1: configuration_T.
```

confTemplate1: configuration_T.

```
confTemplate1: configuration_T.
```

Note that when a parameter type is meant to have sub-types it is expressed with the : relation (has_a relation). This way one may easily consult the configuration database; for instance, ?X:confTemplate1 will retrieve all the sub-types in confTemplate1. More in general, the powerful $\mathcal{F}$ LORA-2 queries make it very easy to deal with CTs, by implementing operations like expanding all possible configurations represented by a template, constructing the union or the intersection of two CTs, and so on. This is very useful for the different uses of CTs described later on.

### 4.4.3 Solution Generation

Given the model has been solved, hence the solvers have returned back the solution objects with the variables and their values, the system will automatically generate a XML file with the processes' result.

```
<STATUS value=" status"/>
<VARIABLES>
    <V n="var_name" v="value" />
</VARIABLES>
<OTHER>
</OTHER>
```

This file is formed by a first tag STATUS, where value describes the exit status of the solution process (at this point we only report if the process succeed in finding a solution or not, of course this must be improved with more detailed information). In case a solution was found, the file will contain a tag VARIABLES with the values of all variables.

This first two tags are generated by the default implementation of the class c_solution. This class also offers an empty extendable method otherinfo () that returns a list of XML nodes. This method may be extended by any c_solution's descendant and the XML Node generated will always be placed inside the tag OTHER.

Note that the only solution object that will be asked to provide the XML file, is the one returned by the solver applied to the root structure in the formulation.

### 4.4.4 The overall solution process

When the SI is generated, the Solvers' Tree assigned and the configurations decided, I-DARE(solve) is ready to start the solution process. This process will commence by the invocation of a Dispatcher
that deals with the Meta Data file in SI, initializes the solution process by creating a tree of solver invocations depending on the solvers' disposition inside the Solvers' Tree.

Note that each solver invocation must be parametrized with the piece of instance it needs and the corresponding configurations. See Figure 4.8


Figure 4.8: Overall solution process

The dispatcher is designed to contribute little overhead to the overall solution process. The overall execution time will be mainly influenced by the running times of the used solvers.

## Chapter 5

## I-DARE(t) - the reformulation system

Abstract
Once a formulation is defined, and the enhanced instance has been generated using the data of the actual instance at hand, we could directly pass this instance to the solvers and obtain the solution. However, there may be equivalent (re)formulations for which a more efficient solution methods could be applied. This chapter will present I-DARE's methodology for creating (and semantically dealing with) these reformulations.

### 5.1 Atomic Reformulation Rules

Reformulations in I-DARE will be done using an approach based on mapping the input and output of the structures involved. If we want to reformulate structure class $A$ into $B$ we need to define a map from A's arguments to B's arguments and another map from B's answer (variables) to A's answer. These mappings will conform a reformulation rule between A and B. The set of all specified reformulation rules, plus their semantics will define I-DARE $(\mathrm{t})$, the reformulation system.

One common point of every structure class (?C::d_Component_C) is that they ultimately need arguments and provide with an answer. Arguments, as defined in §1.1.1, are lists composed of variables, constants, expressions, relations and/or direction. On the other hand, answers are the values taken by the variables that are present in the argument list (these values are assigned by the solvers).

To make a reformulation of a certain model, $\operatorname{I-DARE}(\mathrm{t})$ applies step by step a set of Atomic Reformulation Rules (ARR). Each ARR will deal with a particular structure, transforming it into another equivalent structure. All ARRs must be an instance of the following class,

Listing 5.1: ARR class definition

```
d_ARR
[
    A => d_Component_C,
    B => d_Component_C
].
```

where A and B are the components involved in the reformulation (from A to B).
I-DARE ( t )'s set of ARRs will be divided in two main categories, algebraic and algorithmic. The first one defines the mappings using the constructs offered by I-DARE(im) to build expressions (see 2.2.4). The data will be transformed automatically using implemented handlers (to deal with the reading and writing of the data). To finally solve the problem, the applied ARR will induce the
automatic generation of a $\mathrm{C}++$ solver that will use the transformed input, delegate its solution to a B's solver and transform back the answers to A.

On the other hand, the second one relies on already C++ implemented delegation solvers that will deal with the transformation of the data and answers. This type of reformulation can be seen as delayed, since it is applied only when the instance is being solved. In this case the delegation solver will be implemented in order to convert the input, delegate the solution on another known solver and transform back the answer.

We will not be able to create reformulation rules directly from d_ARR. In fact, in sections to come we will enter in the particular aspects of each type of ARR, explaining how we can effectively create reformulation rules from them. However, we will first need to clarify some aspects common to both types of ARR.

### 5.1.1 Abstract structure classes

ARRs allow us to represent a reformulation rule from one structure class to another. To build an ARR the only information we have is the classes stored in A and B. When dealing with non abstract classes we have all the information we may need about the structure in order to design an ARR. We can access the dMVs, the parameters, the substructure classes, the linking information, we even know all the semantical value enclosed in that structure class.

However, the usage of an abstract structure class or of a block class that contains abstract structure class in it, when declaring A or B inside an ARR, may conduct us to the problem of not having enough information to build that ARR.

For instance, the block class d_B_vardept_C has as substructure class d_Component_C. Having this kind of block classes in A or B's declaration makes it impossible to access the inner substructure parameters while defining the argument and/or answer mappings. Other abstract descendants of d_Component_C may present an issue, for instance d_B_Lagrangian_Relax_C uses an abstract class when defining its substructures (d_LR_C). This class only defines how the equivalent blueprints must be, but has no structural information whatsoever to build an ARR.

Therefore we need a way to avoid the usage of abstract structure classes while defining an ARR, but still being able to build ARRs that involve classes like d_B_Lagrangian_Relax_C or d_B_vardept_C. I-DARE $(\mathrm{t})$ defines a way of narrowing down a structure class that may contain abstract classes.

Definition 5.1.1 (Narrowed Leaf Problem) Let ?pl :: d_LeafProblem_C then we say that ?pl is a narrowed leaf problem, iff ?pl[not abstract]. If ?pl is not narrowed then ?npl :: ?pl, such that ?npl [not abstract], is called a narrowing of ?pl.

A narrowing of a leaf problem (and also of a block) will consider the extension of the :: operator based on the blueprint equivalence rule (see Definition 4.4.1 and Listing 4.4).

Definition 5.1.2 (Narrowed Block) Let ?bl :: d_Block_C, such that ? $\quad$ [not abstract] then we say that ?bl is a narrowed block, iff $\forall$ ?sub $\in$ ?bl.subsC ?sub is narrowed. A narrowing for ?bl is defined by the following term, ?bl(?nsub ${ }_{1}, \ldots$, ? nsub $_{\mid}$?bl.subs ${ }_{\mid}$), such that

$$
?_{\text {nsub }_{i}}= \begin{cases}?- & \text { if ?bl.subsC }[i] \text { is narrowed } \\ \text { a narrowing of ?bl.subsC }[i] & \text { otherwise }\end{cases}
$$

 of the narrowing is the class of its functor).

The narrowing of a substructure can be done using a tuple of different structures classes. For instance, assume we have

- the class d_B_Lagrangian_Relax_C;
- that d_MCF_C_blueprint and d_Simple_Selection_C_blueprint are the blueprints of d_MCF_C and d_Simple_Selection_C, respectively; and
- d_MCF_C_blueprint $\leadsto$ d_LR_C_blueprint and d_Simple_Selection_C_blueprint $~$ d_LR_C_blueprint;
then a narrowing for d_B_Lagrangian_Relax_C could be d_B_Lagrangian_Relax_C((d_MCF_C, d_Simple_Selection_C), ?_).
In this example d_LR_C is narrowed to the tuple (d_MCF_C, d_Simple_Selection_C) and d_Linear_Constraints_C is already narrowed, hence ?- appears in its place. Also note that the variables for d_MCF_C and d_Simple_Selection_C will be created to be disjoint, in order to create a well-formed block (see Definition 2.5.1).

From these previous definitions, we will imply the following rule that must be applied to all ARRs.

Definition 5.1.3 (ARR narrowing rule) Let ?arr:d_ARR, such that ?arr [A -> ?a] and ?arr [B -> ?b], then ?a and ?b must be narrowed structure classes or narrowings.

We can also register a solver to a narrowing, instead of registering it to the original class name. This may become handy, when we do not have a solver for the general not narrowed block, but we do know how to tackle a particular narrowing of that block.

### 5.1.2 Track Structures

When an ARR is applied to a structure inside a formulation, it must leave a track of that application. A mere exchange of structures may seem appropriate, however it is highly problematic. It may create well-formedness problems and incompatibilities when applying the solvers. Moreover, it may be costly to maintain a record of the ARR already applied.

I-DARE ( t$)$ proposes a different policy instead. Each time an ARR is applied to a structure it will leave a trace on that structure name. That trace will be called track structure.

Definition 5.1.4 (Track structure) A track structure is a term with the following form _tr (s, $\left.\mathrm{b}_{1}, \ldots, \mathrm{~b}_{m}\right)$, where $s$ is a structure or a structure class, $m>0, b_{i}$ is a narrowed structure class or a track structure, $\mathrm{s}_{\mathrm{f}} \mathrm{b}_{i}$ and $\mathrm{b}_{i} \neq \mathrm{b}_{j}$, with $i \neq j \in[1 . . m]$.

Note that a track structure will not allow repeated structures within it.
The system will allow the application of certain operations to a track structure. Let _tr ( $\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}$ ) be a track structure then

- _tr $\left(b_{0}, b_{1}, \ldots, b_{m}\right): ? \mathrm{C}$, will be defined as $\mathrm{b}_{0}: ? \mathrm{c}$. The class of the track structure is the class of its first structure.
- tail $\left({ }_{-t r}\left(\mathrm{~b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}\right)\right)=\mathrm{b}_{m}$,
- head $\left(\operatorname{tr}\left(\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}\right)\right)=\mathrm{b}_{0}$,
- rest $\left(\operatorname{tr}\left(\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}\right)\right)=\operatorname{tr}^{\operatorname{tr}}\left(\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m-1}\right)\left(\right.$ or $\mathrm{b}_{0}$ if $\left.m=1\right)$,
- $\operatorname{addS}\left(\operatorname{tr}\left(\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}\right), \mathrm{b}_{m+1}\right)=\operatorname{-tr}\left(\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}, \mathrm{~b}_{m+1}\right)$,
- $\operatorname{tr}\left(\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}\right)[j]=\mathrm{b}_{j}$, with $j \in[0 . . m]$.
- $\operatorname{from}\left(j, \operatorname{tr}^{\operatorname{tr}}\left(\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}\right)\right)=\operatorname{Ltr}^{\left(\mathrm{b}_{j}, \ldots, \mathrm{~b}_{m}\right)}$, with $j \in[1 . . m]$.

- $\forall \operatorname{Method}(\ldots)$, $\operatorname{tr}\left(\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}\right) \cdot \operatorname{Method}(\ldots)=\mathrm{b}_{0} . \operatorname{Method}(\ldots)$. For example $\operatorname{tr}\left(\mathrm{b}_{0}, \mathrm{~b}_{1}, \ldots, \mathrm{~b}_{m}\right)$.freeinds $=$ $\mathrm{b}_{0}$.freeinds

Proposition 5.1.5 (Track structure innocuousness) Given a well-formed formulation F:: d_Formulation that contains a structure s:d_Component_C, then we can substitute s inside of $F$ by any track structure ts, such that head $(t s)=s$, and $F$ will remain well-formed.

Proof. Assume the contrary, F do not remain well-formed. This means that ts is of the wrong structure class, or one method of ts gave an unacceptable value in the well-formedness verification. But, since the class of $t s$ is the same class of $s$, and all ts methods are equal to the $s$ methods, $F$ would have been not well-formed even in its original form (with s inside instead of ts). Then by absurd, F remains well-formed.

Track structures give us the means to store inside a formulation the trace leaved by the application of ARRs. But not all track structure represents a sequence of ARR application, so we need a rule to control the track structures' creation.

The track structures enables us to apply the ARRs in a transitive way, by expanding the track structure.

Definition 5.1.6 (Track structure creation rule) Given that ?arr:d_ARR, such that ?arr [A -> ?ac] and ? arr [B $->$ ?bc], ?F:d_Formulation, ? ts $\in ? \mathrm{~F}$ is a track structure (or a normal structure) such that tail (?ts )::? ac, and ?arr is applicable to tail (?ts), then a new ?nts will be created, to substitute ?ts inside of ?F, such that,

$$
? \mathrm{nts}= \begin{cases}\operatorname{tr}(? \mathrm{ts}, ? \mathrm{bc}) & , \text { if } ? \mathrm{ts} \neq \operatorname{tr}(\ldots) \\ \text { addS(?ts, ?bc) } & , \text {,otherwise }\end{cases}
$$

The track structure creation rule define how track structures are incrementally formed from the application of the ARRs. Note the previous definition uses the concept of applicable ARR, which will be defined in $\S 5.2 .7$ and $\S 5.3 .1$.

### 5.2 Algebraic ARR

As its name already states, an Algebraic ARR (ARR ${ }^{\Sigma}$ ) uses only algebraic constructs to perform both mappings. To build these algebraic constructs we will need to access the values (of constants and variables), the cardinality of the dimensions, the existing relations and direction. But when building an $\mathrm{ARR}^{\Sigma}$ the only information we have in our hands is the structure classes A and B. We then need a way to access all the data inside those structure classes.

For that purpose I-DARE $(\mathrm{t})$ defines the binary operator A..f (extraction operator).
Definition 5.2.1 (Extraction Operator) Let ?A::d_Component_C be any non abstract structure class, then the extraction operator ?A..f is defined as follows,

- if ?A::d_LeafProblem_C and $f$ is an ?id such that (?id, ?type) $\in$ ?A.args, then
- if ?type is d_var, d_constant, d_rel or d_direction then A..f represents the value of that variable, constant, relation or direction, respectively.
- if ?type $=$ d_vector(?K, ?S) and assuming we have one index $i_{j}$ for each ?S[j], $j \in[0 . \mid$ ?S| -1$]$ then ?A..f( $\left.\mathrm{i}_{0}, \ldots, \mathrm{i}_{\mid} \mathrm{PS}_{\mid-1}\right)$ represents the value of the vector in the specified position.
- if ?A::d_LeafProblem_C and $f$ is a ?dimvar $\in$ ?A.dim_var then ?A..f represents that dimension's meta variable.
- if ?A::d_Block and $f$ is an ?id $\in$ ?A.ids, then ?A..f represents the substructure ?A.subsC[?id].

Observe that if we have a block class ?A that has a substructure with identifier id, then we can apply the extraction operator on that substructure by doing ?A..id .. f. This way we can access a whole block structure.
$\operatorname{ARR}^{\Sigma}$ will use the extraction operator applied to the properties A and B. For instance, assume we have an $A R R^{\Sigma}$ for which A -> d_LP_C, then we could do the following extraction operations,

- A..cols retrieves one of the dimensions of d_LP_C to which the cardinality operator can be applied, _card (A.. cols);
- A.. dir retrieves the direction of d_LP_C;
- A..c(i), with i an index over cols, retrieves the values of the price constants of the objective function of d_LP_C.

The extraction operator plays an important role in the definition of an $\mathrm{ARR}^{\Sigma}$. Let's now define how $\mathrm{ARR}^{\sum}$ can be created as an object derived from the following class,

Listing 5.2: $\mathrm{ARR}^{\Sigma}$ class definition

```
d_ARR_Algebraic::d_ARR
[
    indexA => _list ,
    indexB => _list,
    dimRel => _list,
    arg_map => _list ,
    ans_map => _list,
    fixTI => _list, // only if B::d_Block_C
    [condition => _term]
].
```

where

- indexA and index - are lists of $\operatorname{ARR}^{\Sigma}$ index declarations (see $\S 5.2 .1$ ),
- dimRel - is a list of $\mathrm{ARR}^{\Sigma}$ dimension relations (see §5.2.2),
- arg_map - is a list of $\mathrm{ARR}^{\Sigma}$ argument mappings (see §5.2.3),
- ans_map - is a list of $\mathrm{ARR}^{\Sigma}$ answer mappings (see §5.2.3),
- fixTI - is a list of fixed template items (see §5.2.4), that must only be defined if B::d_Block_C,
- condition - is an optional conditional expression that must be satisfied for the $\operatorname{ARR}^{\sum}$ to be used.


### 5.2.1 $\mathrm{ARR}^{\Sigma}-$ Index declarations

While defining the extraction operator we made reference to declared indices. But since the ARR ${ }^{\Sigma}$ is working at a structure class level, there are no index declared. Therefore, we need to specify those indices inside the $A R R^{\Sigma}$. For that purpose are the index declarations indexA and indexB.
indexA will be a list containing elements of the form $? d=\left(i_{0}, \ldots, i_{m}\right)$, where $A .$. ?d is a valid extraction operation of a dMV and $i_{j}$ is a unique identifier in indexA that will represent an index for dimension $A$..?d. If $m=1$ then we can suppress the parenthesis. For indexB the specification is analogous, but using structure class B instead.

For example, if A were d_LP_C then we could create the following index declaration, indexA $->[$ cols=( $\mathrm{i}, \mathrm{j})$, cons=k].
When the structure in $A$ is a block that has a substructure s such that A. freeinds $\cap$ s.freeinds $\neq \emptyset$, then $s$ will be replicated inside of $A$. This situation implies that we would want to access each replication of s independently. At this point, using dMV to declare the indices may not be enough. In fact, we will need to declare an index over A. freeinds $\cap$ s.freeinds.

Definition 5.2.2 (Free Index Difference) Let A be a block and s a substructure of A, then the free index difference of s in A , is denoted by $-\mathrm{fiD}(\mathrm{s})=\| \mathrm{A}$.freeinds $\cap \mathrm{s}$.freeinds $\|$.

Note that due to block well-formedness (cf. Definition 2.5.1) the A. freeinds $\subseteq$ s.freeinds. Therefore A. freeinds $\cap \mathrm{s}$.freeinds $\subseteq$ s.freeinds.

Then ?d besides being a dMV it may be _fiD(s), if $s$ is a substructure of $A$ (or $B$ if we are defining indexB). Of course this can only occur when A (or B ) is a block.

### 5.2.2 $\mathrm{ARR}^{\Sigma}$ - Dimension relations

When reformulating one structure into another, it is imperative to know how the dimensions of both structures are related between each other. In the case of $\mathrm{ARR}^{\Sigma}$ this relation will be through an expression that involves as operators: dMVs and numeric constants.

This relations between dimensions will be contained in the dimRel list. Each element of dimRel will have the form, ?d=expr, where B..?d is a valid extraction operation of a dMV, and expr is an expression that uses as operands: A..?d1 (a valid extraction operation of a dMV), free index difference operations and/or numeric constants.

For instance assume we are reformulating the d_LP_C into d_LP_MPS_C. In this case the dimension relation must be dimRel $->$ [cols=cols, cons=cons]. More complex example will be seen later on.

The dimension relation list of an $\operatorname{ARR}^{\Sigma}$ must guarantee that all dMV of the structure class B are computed inside dimRel. For formalizing such a rule, we need to define how to extract all dMVs from any structure class (even from blocks).
Definition 5.2.3 (dMV Join for Leaf Problems) Let ?P::d_LeafProblem_C, then the $d M V$ Join of ?P, will be defined as ?P.dmv_join $=$ ?P.dim_var.

Definition 5.2.4 (dMV Join for Blocks) Let ?B::d_Block_C, then the $d M V$ Join of ?B, will be defined as

$$
\text { ?B.dmv_join }=\{? \text { ?id..f|?id } \in \text { ?B.ids } \wedge \mathrm{f} \in \text { ?B.subsC[?id].dmv_join }\}
$$

In the previous definition we used the operator .. to access a parameter of even a substructure inside a block. When a block has a substructure composed by a tuple of structures, for instance, B -> d_B_Lagrangian_Relax_C((d_MCF_C, d_Simple_Selection_C), ?-)), we can access the parameters of the elements of the tuple individually by doing B..sub(d_MCF_C) for the first element of the tuple and B..sub(d_Simple_Selection_C) for the second one.

The following definition declares a rule that all $\mathrm{ARR}^{\Sigma}$ must satisfy.
Definition 5.2.5 (Dimension Relation Rule) Let ? arr be an $\operatorname{ARR}^{\sum}$, then $\forall$ ?d $\in$ ?arr.B.dmv_join there must be a ?d=expr $\in$ ?arr.dimRel.

### 5.2.3 $\mathrm{ARR}^{\Sigma}-$ Mappings

The argument mapping defines how the parameter of one structure are transformed into the other's. The main idea resides in specifying for each parameter of B how it will be built depending on A's parameters. Of course this depends on the type of the B parameter we are trying to compute.

First we will need to describe how the expressions used for mapping will be constructed. As in the case of dimensions, mapping expressions will use a particular set of operands, which are: extraction operations (e.g. A..c(i,j)), numeric constants, relations ( $=<,>=,=$ ) and/or directions (min, max).

For instance, if are trying to compute a constant, the following expression could be valid for doing so, $\mathrm{A} . . \mathrm{c}(\mathrm{i}) * 2+\operatorname{card}(\mathrm{A} . . \mathrm{cols})$, if $\mathrm{A} \rightarrow$ d_LP_C. $^{\text {C }}$

When $A$ is a block and $s$ is a substructure of $A$, such that A. freeinds $\cap$ s.freeinds $\neq \emptyset$, then we will be able to do the following A..s(f), where f is an index over _fiD(s). Note that this is a way of accessing a particular replication of $s$.

Now let's define, for each type of parameter, how the mappings must be specified.
Definition 5.2.6 (Scalar Mapping) Let B..f be of scalar type, then a scalar mapping must be done, and it is defined as follows, B.. $f=$ ?MP, where

- if ( $f=$ d_constant $) \in B$.args then ?MP is a constant expression involving the following operands: extraction operations of type d_constant, numeric constants, cardinality operators (e.g. card(A..cols)), and/or directions (seen as their numeric values ( $1,-1$ )).
 or a relation constant $(=<,>=$ or $=)$.
- if ( $\mathrm{f}=$ d_direction ) $\in$ B.args then ?MP must be exactly one extraction operation of type d_direction or a direction constant (min or max).

Definition 5.2.7 (Vector Mapping) Let B..f be of type d_vector(?K, ?S), with ?K $\neq$ d_var, then a vector mapping must be done, and it is defined as follows, B..f $=$ ? MP, where

- ?MP is a vector construction (see §2.3), such that ?MP has |?S| dimensions, and ?MP is built using exclusively scalar mappings corresponding to type ?K; or
- ?MP is a extraction operation of type d_vector(?K, ?S1), such that |?S| = |?S1|.


## Variable partitioning

Sometimes, mainly when reformulating into a block, we need to partition the variables of a certain substructure. For instance, assume the B structure of an ARR ${ }^{\Sigma}$ is d_B_MILP_C,

```
d_B_MILP_C :: d_Block_C
[
    ids -> [ Im, ic, d_loc(d_IC_C)],
    link -> [([X,Y],d_all), ([X],d_all)]
].
```

Note that in this case the variables of d_LP_C must be partitioned in order to unify with the template pattern $[\mathrm{X}, \mathrm{Y}]$. However d_LP_C has a single way to access its variables, that is through the property $x$. Hence, we have no way of partitioning $x$ to unify with $[\mathrm{X}, \mathrm{Y}]$ with the tools so far exposed. For this reason we define the variable partitioning.

Definition 5.2.8 (Variable Partitioning) Let B..f be of type d_vector(d_var, ?S), then a variable partitioning is defined by, B..f $=? \mathrm{~L}$, where ? L is a list composed by elements of the form, $\mathrm{n}=? \mathrm{D}$, with

- $n$ being a atom, unique inside of the reformulation where resides the partitioning; and
- ? $\mathrm{D}=\left[\mathrm{D}_{1}, \ldots, \mathrm{D}_{|? \mathrm{~S}|}\right]$, such that $\mathrm{D}_{i} \in \mathrm{~A} . d m v \_j$ join, $i \in[1 . .|\mathrm{S}|]$ or is a free index difference; or
- ? $\mathrm{D}=\quad \operatorname{rrpl}\left(\mathrm{D}^{\prime}, \mathrm{L}\right)$, where $\mathrm{D}^{\prime} \in \mathrm{A} . d m \mathrm{~d}$ _join or is a free index difference, and $\mathrm{L}=\left[\mathrm{D}_{1}, \ldots, \mathrm{D}_{|? \mathrm{~S}|}\right]$, such that $\mathrm{D}_{i} \in \mathrm{~A}^{\text {.dmv_join, }} i \in[1 . .|\mathrm{S}|]$ or is a free index difference.

The definition of variable partitioning allow us to access a subset of the variables in a structure. For instance, if we have that $\mathrm{B} . . \mathrm{x}=[\times 1=[\ldots], \times 2=[\ldots]]$, we use the terms B... $\times(\times 1)$ or $\mathrm{B} . . \times(\times 2)$ to access the partitions in $\times$.

The usage of $\quad$ rpl $\left(D^{\prime}, L\right)$ creates $\left|D^{\prime}\right|$ partitions of size $\|L\|$, so the accessing term will be a little more complex. Assume, we have $\mathrm{B} . . \mathrm{x}=\left[\mathrm{xp}=\_\operatorname{rpl}(\mathrm{D},[\mathrm{D} 1, \mathrm{D} 2])\right]$ and that we have indices declared over D1 and D2, being $i, j$ respectively; then we can use the terms B...x(xp(i,j)) to access the partitions in x .

## Shared indices

There is a particular case of reformulation in which we will transform a block, with shared variables, into another structure. For instance, assume we have a block C with substructures s1, s2 and link $[[X, Y],[X, Z]]$. Note that C..s1 shares the variables unified to $X$ with C..s2. Now, if we try to reformulate C into a structure S , we may need to access the indices involved with $\mathrm{X}, \mathrm{Y}$ or Z , and even more in the case of $X$ (that is shared) we may want to access the indices of $X$ contextualized in C..s1 and contextualized in C..s2, independently.

Let us define a way to retrieve all shared and non-shared template items in a block's link.
Definition 5.2.9 (Shared Template Items) Given a block C of class cC such that cC. link $=\left[\mathrm{TP}_{1}, \ldots, \mathrm{TP}_{k}\right]$ and C.subs $=\left[\mathrm{s}_{1}, \ldots, \mathrm{~s}_{k}\right]$, then the shared template items of C is denoted by the following set:
C.sharedTI $=\left\{\left[\mathrm{X},\left[\mathrm{s}_{i_{1}}, \ldots, \mathrm{~s}_{i_{m}}\right]\right] \mid \mathrm{X} \in \operatorname{TP}_{i_{1}} \wedge \ldots \wedge \mathrm{X} \in \operatorname{TP}_{i_{m}} \wedge i_{j} \in[1 . . k] \wedge j \in[1 . . m] \wedge m\right.$ is as big as possible $\}$.

Note that the fact of assuming $m$ as big as possible ensures that if there is a shared template item it will include all the structures where the template item appears. For instance, if we have the link $[[X, Y],[X, Z],[Y, W]]$ and the substructures $s_{1}, s_{2}, s_{3}$ in a block $C$ then

$$
\text { C.sharedTI }=\left\{\left[\mathrm{X},\left[\mathrm{~s}_{1}, \mathrm{~s}_{2}\right]\right],\left[\mathrm{Y},\left[\mathrm{~s}_{1}, \mathrm{~s}_{3}\right]\right],\left[\mathrm{Z},\left[\mathrm{~s}_{2}\right]\right],\left[\mathrm{W},\left[\mathrm{~s}_{3}\right]\right]\right\}
$$

To identify the set of instanced indices associated to a particular linking variable (unified with a template item) we will use the operator template item dimension, defined as follows,

Definition 5.2.10 (Template Item Dimension) Given a block C such that C.subs $=\left[s_{1}, \ldots, s_{k}\right]$ then the template item dimension operator can be defined as $\operatorname{tiD}\left(\mathrm{X},\left[\mathrm{s}_{i_{1}} . . v_{i_{1}}, \ldots, \mathrm{~s}_{i_{m}} . . \mathrm{v}_{i_{m}}\right]\right)$, with $\left[\mathrm{X},\left[\mathrm{s}_{i_{1}}, \ldots, \mathrm{~s}_{i_{m}}\right]\right] \in \mathrm{C}$. sharedTIUC.nsharedTI and $\mathrm{v}_{i_{j}}$ is a variable identifier in $\mathrm{C} . . \mathrm{s}_{i_{j}}$, with $i_{j} \in[1 . . k]$ and $j \in[1 . . m]$; and it will represent the set of indexing values associated with the template item and structure(s) specified.

Note that _tiD (...) represents a dimension that has a size and indices can be declared over it (like any other dimension). Observe that if the index is declared in indexA then A will replace C in the previous definition. Moreover, _tiD (...) can be used inside the variable partition declaration (as a dimension).

A dimension of form _tiD (X,L) will define two possible sets depending on the way of usage. The first one is defined by the instanced indices of the variable $\mathrm{L}_{i}$ unified with X , this set will be called shared instanced indices, and its defined as follows,

Definition 5.2.11 (Shared Instanced indices) Given a block C with a substructure shaving a variable $v$, with indices $I_{v}$, such that $v$ is shared through template item X but only for the instanced indices $s I \subseteq I_{v}^{\xi}$, then $s I$ defines the shared instanced indices of $v$ in sthrough X , and it is denoted by $I_{v}^{\xi(\mathrm{X}, \mathrm{s})}$.

In the second case of _tiD (...) usage, it will simply define the set $\left[0 . . \| \_\right.$tiD $\left.(\ldots) \|-1\right]$. This cases of _tiD (...) usage are determined by how the indices declared over _tiD (...) are utilized. The first case arises only the index is used inside a structure that appears in _tiD (...). The second case is applied otherwise. This indices defined over _tiD (...) are called shared indices. Let us formally define the shared indices and both cases of usage.

Definition 5.2.12 (Shared index) Given an index i declared over _tiD ( X, ? L ) it will be called shared index, if i is used inside a term of the form C..s.. $\mathrm{f}\left(\mathrm{j}_{1}, \ldots, \mathrm{j}_{l-1}, \mathrm{i}, \mathrm{j}_{l+1}, \ldots, \mathrm{j}_{m}\right)$, where

- C is either A or B;
- $s$ can a term of the form $s_{1} \ldots \ldots s_{k}, s_{1} \in C . s u b s, s_{2} \in s_{1}$.subs, and so on;
- f is a parameter of $\mathrm{s}_{k}$;
- $\mathrm{D}_{1}, \ldots, \mathrm{D}_{m}$ are the dimensions of f ;
- C..s..v $\in$ ? L and $I_{v}^{\xi(\mathrm{X}, \mathrm{s})}=\left\|\mathrm{D}_{l}\right\|$,
then i will take values on the set $I_{v}^{\xi(\mathrm{X}, \mathrm{s})}$, otherwise i will take values in the set $[0 . .\|\operatorname{ltiD}(\mathrm{X}, ? \mathrm{~L})\|-1]$.

The shared indices can be used to access the actual variable' indices that were shared through a template item, but depending on the structure the template item is used on.

## Bound Mapping

When reformulating A into B we may need to express the variable bounds of $B$ (i.e. the lower and upper bounds of the variables in $B$ ). To achieve this we will use a construct named bound mapping.

Definition 5.2.13 (Bound Mapping) Let B..f be of type d_var or d_vector(d_var,?S), then a bound mapping may be done, and it is defined as follows, $\mathrm{B} . . \mathrm{f}=$ ? bm(?MP), where

- ?bm must be lower or upper, depending on which bounding we are declaring,
- ?MP must be a
- scalar mapping of type d_constant if B..f is of type d_var
- vector mapping of type d_vector(d_constant, ?S).

Using the scalar, vector and bound mappings, we will define how the entire argument mapping can be done. However, to define the argument mapping we will need a function like the one defined for extracting dMVs, but in this case for extracting all parameters (also for block classes). The parameters will be divided into variable and non-variable type parameter, therefore we will use to functions, vparam_join and param_join.

## Argument and Answer Mappings

To make simpler the definitions of the argument and answer mappings, we will build the sets of non-variable and variable parameters of a structure.

Definition 5.2.14 (non-variable Parameter Join for Leaf Problems) Let ?P::d_LeafProblem_C, then the non-variable Parameter Join of ?P, will be defined as

$$
\text { ?P.param_join }=\{? \text { id } \mid(? \text { id, ?type }) \in \text { ?P.args } \wedge \text { ?type } \neq \text { d_var } \wedge \text { ?type } \neq \text { d_vector(d_var, ?S })\} .
$$

Definition 5.2.15 (variable Parameter Join for Leaf Problems) Let ?P::d_LeafProblem_C, then the variable Parameter Join of ?P, will be defined as

$$
\text { ?P.vparam_join }=\{? \text { id } \mid(? \text { id, ?type }) \in \text { ?P.args } \wedge(\text { ?type=d_var } \vee \text { ?type=d_vector }(\text { d_var, ?S }))\}
$$

Definition 5.2.16 (non variable Parameter Join for Blocks) Let ?B::d_Block_C, then the nonvariable Parameter Join of ?B, will be defined as

$$
\text { ?B. param_join }=\{\text { ?id..f|?id } \in \text { ?B.ids } \wedge \mathrm{f} \in \text { ?B.subsC[?id].param_join }\} \text {. }
$$

Definition 5.2.17 (variable Parameter Join for Blocks) Let ?B::d_Block_C, then the variable Parameter Join of ?B, will be defined as

$$
\text { ?B.vparam_join }=\{\text { ?id..f|?id } \in \text { ?B.ids } \wedge \mathrm{f} \in \text { ?B.subsC[?id].vparam_join }\} .
$$

Definition 5.2.18 (Argument Mapping) Let ?arr be an $\mathrm{ARR}^{\Sigma}$ then the argument mapping must have the following form, arg_map -> ?L, with ?L being a list composed of the following terms,

- for all ? id $\in$ ? arr.B.param_join, one must add (B..? id = ?ME) to ?L, where ?ME is a scalar or vector mapping depending on ?id's type; and
- Optionally one may add ( $\mathrm{B} .$. ?id $=$ ? lu ) to ? L , where ? id $\in$ ? arr. B.vparam_join and ?lu is a bound mapping.
- Optionally one may add ( $\mathrm{B} .$. ? id $=$ ?vp) to ? L , where ? id $\in$ ? arr. B.vparam_join, ?id is a d_vector(d_var, $\left[\mathrm{V}_{1}, \ldots, \mathrm{~V}_{m}\right]$ ) and ? $\mathrm{vp}=\left[\mathrm{n}_{1}=\mathrm{P}^{1}, \ldots, \mathrm{n}_{k}=\mathrm{P}^{k}\right]$ is a variable partitioning, such that,

$$
\left\|\mathrm{V}_{i}\right\|=\sum_{j \in[1 . . k]}\left(\left\{\begin{array}{ll}
\mathrm{D}_{i}^{j} & \text { if } \mathrm{P}^{j}=\left[\mathrm{D}_{1}^{j}, \ldots, \mathrm{D}_{m}^{j}\right] \\
\mathrm{D}^{*} \mathrm{D}_{i}^{j} & \text { if } \mathrm{P}^{j}=\operatorname{rrp}\left(\mathrm{D},\left[\mathrm{D}_{1}^{j}, \ldots, \mathrm{D}_{m}^{j}\right]\right)
\end{array}\right)\right.
$$

$\forall i \in[1 . . m]$.

Note the arg_map must define a mapping for all the B's parameters that are not of variable type. On the other hand for those parameter of variable type, arg_map may optionally define a bound mapping.

The answer mapping is very similar to the argument mapping. The answer mapping transforms the variable values of B into the variable values of A . So, there are no constants, relations or directions, just variables. Since the answer mapping is applied once the structure B is solved, the variables of B will have numeric values assigned.

Definition 5.2.19 (Answer Mapping) Let ?arr be an $\mathrm{ARR}^{\Sigma}$ then the answer mapping must have the following form, ans_map -> ?L, with ?L being a list composed of the following terms,

- for all ?id $\in$ ? arr.A.vparam_join, one must add (A..?id $=$ ?ME) to ?L, where ?ME is a scalar or vector mapping of constant type,
?id may be optionally indexed if needed (using the indices defined in indexA).
Note that while building the answer mapping in an $\mathrm{ARR}^{\Sigma}$, the variables of B will be considered constants, because the solution of B was already obtained at this point, hence all variables has a constant value assigned.


### 5.2.4 $\mathrm{ARR}^{\sum}$ - Fixed template items

When B::d_Block_C we need to define how the B. link will be constructed. For doing so, ARR ${ }^{\sum}$ uses the method fix $\mathrm{TI}->$ ? L, specifying which variable tuple will correspond to each template item in B. link (see Definition 1.2.2 and 1.2.6).

Assume we have a method get_ti that extracts the list of all template items from a block's link. For example, if we have the block,

```
d_B_vardept_C :: d_Block_C
[
    ids -> [master , slave ],
    subsC }->\mathrm{ [d_Component_C, d_Component_C],
    link -> [[X,V] , [X,W]
].
```

then d_B_vardept_C. get_ti will return the list [ $\mathrm{X}, \mathrm{V}, \mathrm{W}$ ].
Definition 5.2.20 (Fixed template items) Let ? arr be an ARR ${ }^{\Sigma}$, such that ?arr.B :: d_Block_C, then the fixed template items' list must be specified in ?arr, and it is defined as fix $\mathrm{TI}->$ ? L , such that $\forall($ ?ti $\in$ ?arr.B.get_ti) ?ti $=$ B..?sub..?v must belong to ?L, having that ?sub..?v $\in$ ?arr.B.vparam_join and ?ti $\in$ ?arr.B.link[?sub].

The fixed template items list specification is mandatory only when the B structure class is a block. An example will be see in $\S 5.2 .6$.

### 5.2.5 $\operatorname{ARR}^{\Sigma}$ - Conditional expression

The conditional expression, to be used in an $\mathrm{ARR}^{\Sigma}$, is a normal condition (as defined in 2.2.4). The only thing that changes is the set of operands. In this case the set of operands adds the following terms,

1. Extraction operations (e.g. A..f), and

## 2. Parent operations.

A parent operation (_parent/1) is very useful to contextualize the structures inside the structure tree of a formulation. In order to define _parent/1 formally, we first need to define the substructure join of a certain structure.

Definition 5.2.21 (substructure Join for Leaf Problems) Let ?P::d_LeafProblem_C, then the substructure join of ?P, will be defined as ?P.subs_join $=[]$.

Definition 5.2.22 (substructure Join for Blocks) Let ?B::d_Block_C, then the substructure join of ?B, will be defined as

$$
\text { ?B.subs_join }=\text { ?B.subs } \cup\left(\cup_{(s b \in ? \text { B.subs })} s b . \text { subs_join }\right) .
$$

subs_join gathers all members of the substructure tree of a certain structure. Notice that root. subs_join will return all structures in a formulation.

Definition 5.2.23 (Parent Operator) Let $F$ be a formulation and $B$ be a structure inside of $F$, then _parent (B) will return any structure class cP, such that, $B \in P$.subs_join and P:cP. If $B=F$.root then _parent (B) will return no structure.

### 5.2.6 $\mathrm{ARR}^{\Sigma}$ - examples

A Minimum Cost Flow (MCF) problem has a LP representation. Assume we have a graph $G(N, E)$, a cost function $\operatorname{cost}_{i j}$, with $(i, j) \in E$, a vector $S D$ such that $S D_{i}>0$, if $i \in N$ is a supply node, $S D_{i}<0$, if $i$ is a destination node, and $S D_{i}=0$ otherwise, and a capacity function $u_{i j}$ for each arc; then we could build the following LP to model the MCF problem,

$$
\begin{array}{ll}
\min & \sum_{(i, j) \in E} \operatorname{cost}_{i j} * \text { flow }_{i j} \\
\text { s.t. } \\
& \sum_{j \in N^{+}(i)} \text { flow }_{i j}-\sum_{j \in N^{-}(i)} \text { flow }_{j i}=S D_{i} \quad i \in N \\
& 0 \leq \text { flow }_{i j} \leq u_{i j} \quad(i, j) \in E
\end{array}
$$

Using the d_MCF_C and d_LP_C classes described in §1.2.1, we can define the following $\operatorname{ARR}^{\Sigma}$, Listing 5.3: $\mathrm{ARR}^{\Sigma}$ to reformulate MCF to LP

```
d_MCF_to_LP_ARR : d_ARR_Algebraic
[
    A \(\rightarrow\) d_MCF_C,
    B \(\quad \rightarrow\) d_LP_C,
    indexA \(\rightarrow[\mathrm{E}=\mathrm{e}, \mathrm{N}=\mathrm{n}]\),
    indexB \(\rightarrow\) [].
    dimRel \(\rightarrow\) [cols=E, cons \(=N\) ]
    arg_map \(\rightarrow\) [
                        B.. dir \(=\min\),
                        B..c \(=\) A..cost ,
                        \(B \ldots A=\$(-\operatorname{cs}([1->(A . . S N(e)=n),-1->(A . . E N(e)=n), 0]), \quad[n, e])\),
            B..rels \(=\$\left({ }^{\prime}=\right.\) ', \(\left.[\mathrm{n}]\right)\),
            \(B \ldots b=\$(A . . S D(n),[n])\),
            B.. \(x=\) lower (0) ,
            B.. \(\mathrm{x}=\operatorname{upper}(\mathrm{A} . \mathrm{u} \mathrm{u})\)
            ],
            A. . flow \(=B . . x\)
            ]
].
```

The ARR ${ }^{{ }^{2}}{ }_{\text {d_MCF_to_LP_ARR, maps }}$ each argument of the d_MCF_C class into the arguments of d_LP_C, and the answers of d_LP_C into the answers of d_MCF_C. Note for argument B..c, that even if it is a vector, we are using a scalar-like notation. This can be done because B..c has the same cardinality as A..cost (see Definition 5.2.7).

This previous example comprises a reformulation between two leaf problems. The next example will describe a $\mathrm{ARR}^{\Sigma}$ between a leaf problem an a block.

Let's consider the Multicommodity Minimum Cost Flow with Fixed-Charge (MMCF(FC)) structure class (already mentioned in §1.3),

Listing 5.4: MMCF FC class

```
d_MMCF_FC_C :: d_LeafProblem_C
[
    dim_var -> [N,E,K],
    args -> [
                SN = d_vector(d_constant, [E]), // start node
                EN = d_vector(d_constant, [E]), // end node
                c = d_vector(d_constant, [K,E]), // cost per unit per arc
                f = d_vector(d_constant, [E]), // fixed cost per design arc
                SD = d_vector(d_constant, [K,N]), // supLfPy/demand
                b = d_vector(d_constant, [K,E]), // single k capacity per arc
                u = d_vector(d_constant, [E]), // mutual arc capacity
                flow = d_vector(d_var, [K,E]), // flow variables
                desg = d_vector(d_var, [E]) // design variables
            ],
    dim_bound -> [(SN,N), (EN,N)]
].
```

Having the following MILP representation

$$
\begin{align*}
& \min \sum_{k \in K} \sum_{(i, j) \in E} \mathrm{c}_{i j}^{k} * \mathrm{flow}_{i j}^{k}+\sum_{(i, j) \in E} \mathrm{f}_{i j} \text { desg }_{i j}  \tag{5.2.1}\\
& \text { s.t. } \\
& \sum_{j \in N^{+}(i)} \text { flow }_{i j}^{k}-\sum_{j \in N^{-}(i)} \text { flow }_{j i}^{k}=S D_{i}^{k} \quad i \in N, k \in K  \tag{5.2.2}\\
& 0 \leq \text { flow }_{i j}^{k} \leq \mathrm{b}_{i j}^{k} \quad(i, j) \in E, k \in K  \tag{5.2.3}\\
& \sum_{k \in K} \text { flow }_{i j}^{k} \leq \mathrm{u}_{i j} \quad(i, j) \in E \quad\left(\gamma_{i j}\right)  \tag{5.2.4}\\
& \sum_{k \in K} \text { flow }_{i j}^{k} \leq \mathrm{u}_{i j} \text { desg }_{i j} \quad(i, j) \in E \quad\left(\alpha_{i j}\right)  \tag{5.2.5}\\
& \text { flow }_{i j}^{k} \leq \mathrm{b}_{i j}^{k} \operatorname{desg}_{i j} \quad(i, j) \in E, k \in K \quad\left(\beta_{i j}^{k}\right)  \tag{5.2.6}\\
& 0 \leq \operatorname{desg}_{i j} \leq 1 \quad(i, j) \in E  \tag{5.2.7}\\
& \operatorname{desg}_{i j} \text { integer } \quad(i, j) \in E \tag{5.2.8}
\end{align*}
$$

From this MILP representation we can construct a Lagrangian Relaxation by dualizing constraints (5.2.4) to (5.2.6). The resulting Lagrangian dual is:

$$
L D=\max _{\gamma, \alpha, \beta \geq 0}\left\{-\sum_{(i, j) \in E} \gamma_{i j} \mathbf{u}_{i j}+L(B 1)+L(B 2)\right\}
$$

where $L(B 1)$ is defined as,

$$
L(B 1)=\min \sum_{k \in K} \sum_{(i, j) \in E}\left(c_{i j}^{k}+\gamma_{i j}+\alpha_{i j}+\beta_{i j}^{k}\right) \text { flow }_{i j}^{k}
$$

subject to constraints (5.2.2) and (5.2.3), and $L(B 2)$ is defined as

$$
L(B 2)=\min _{\operatorname{desg} \in\{0,1\}|E|} \sum_{(i, j) \in E}\left(\mathrm{f}_{i j}-\alpha_{i j} \mathbf{u}_{i j}-\sum_{k \in K} \beta_{i j}^{k} \mathrm{~b}_{i j}^{k}\right) \operatorname{des}_{i j}
$$

The Lagrangian subproblem, therefore, decomposes into $|K|$ MCFs problems and one problem solvable by inspection. Notice that the $L(B 1)$ subproblems can be represented by the structure class d_MCF_C and the $L(B 2)$ by d_Simple_Selection_C. Hence, we could build an $\mathrm{ARR}^{\Sigma}$ to reformulate the d_MMCF_FC_C class into a Lagrangian decomposition d_B_Lagrangian_Relax_C.

Listing 5.5: $\mathrm{ARR}^{\Sigma}$ to reformulate MMCF FC to Lagrangian Flow Relaxation

```
d_MMCF_FC_to_Lagrangian_Flow_Relax_ARR : d_ARR_Algebraic
d
    A -> d_MMCF_FC_C,
    B -> d_B_Lagrangian_Relax_C((d_MCF_C, d_Simple_Selection_C), ?_),
    indexA ->> [E=(e,e1),K=(k,k1),N=n],
    indexB -> []
    dimRel }->\mathrm{ [sub(d_MCF_C).. N=N, sub(d_MCF_C).. E=E,
                sub(d_Simple_Selection_C)..d1=E, linking..d1=K*E+E,
                linking..d2=2*E+E*K],
    arg_map -> [
        //-_ MCF part
        // Note that k is left untouched. Therefore k will be
        // the free-index on which the MCF structure will be replicated.
        B..sub(d_MCF_C)..SN = A..SN,
        B..sub(d_MCF_C)..EN = A..EN
        B\ldotssub(d_MCF_C)..SD = $(A..SD(k,n), [n]),
        B\ldotssub(d_MCF_C)\ldotscost = $(A..c(k,e), [e])),
        B..sub(d_MCF_C)..u = $(A..b(k,e), [e]),
        //-_ Simple Selection part
        B..sub(d_Simple_Selection_C)..f = A..f,
        //-Linear Constraints part (linking constraints)
            B..linking..A = $([ //left hand matrix
                            [ // mutual arc capacity constraints \gamma (ij
                                    $(0-cs([1-> (e=e1), 0]), [e1, (e,k)]),
            ],
                '// mutual arc capacity redundant constraints \alpha \alphaij
                                    $(_cs([1-> (e=e1), 0]), [e1, (e,k)]),
                                    $(-cs([A..u(d1)->(e=e1), 0]), [e1, e] )
                    ],
                            [' // single capacity constraints }\mp@subsup{\beta}{ij}{k
                                $(_cs([1->(e=e1,k=k1), 0]), [(e1,k1), (e,k)])
                                $(-cs([[A..b(k,e)->(e=e1)]), [(e1,k), e]), )
                            1)
                ]),
            B..linking..rels = $([ // relations' vector
                $('=<', [e] ),
                $('=<', [e] ),
                    $('=<', [(e,k)])
            B..linking..b ])$([ // right hand
                        b=$([ // right hand side vector
                            ])
    ans_map -> [
            A.flow = B..sub(d_MCF_C)..flow,
                A.desg = B..sub(d_Simple_Selection_C)..f
    fixTI > [,
        ] X = (B..sub(d_MCF_C)..flow, B..sub(d_Simple_Selection_C )..f)
].
```

Note that in this example we needed to specify a narrowing for $B$ and since $B$ is a block we declared a fixed template items' list. Observe also that the structure sub of the d_Lagrangian_Relax_C block is actually a tuple of structure classes. So the block will have as substructures the |d_MCF_C| and the d_Simple_Selection_C (assuming they have blueprints equivalent to d_LR_C's blueprint).

More complex examples will be seen in the following chapters.

### 5.2.7 Semantics

The previous sections dealt with the syntax of the $\operatorname{ARR}^{\Sigma_{\mathrm{S}}}$, defining how we can declare one atomic reformulation rule, to transform one structure class into another. This section will explain how

I-DARE ( t$)$ deals with the $\mathrm{ARR}^{\Sigma_{S}}$ from a semantic point of view (i.e. what happens when we want to apply an $A R R^{\sum}$ ?).

Now assume we have an ? arr:d_ARR_Algebraic, such that ? arr [A -> ?ac] and ?arr [B -> ?bc] and that we want to apply that $\operatorname{ARR}^{\Sigma}$ to reformulate a structure ?s:?ac into a structure of class ?bc.

First of all we need to transform the arguments from ?ac's format into ?bc format, this way the I-DARE(ei) while generates the structured instance, it will contain the data of ?bc. The solvers will tackle ?bc, and the answer (variable values) must be transform back to ?ac. This process describes, broadly speaking, how I-DARE(t) will deal with the reformulation.

This process mentioned three key concepts in $\mathrm{ARR}^{\Sigma}$ semantics, the first one is the ability to read ?ac and then write ?bc; the second one is the usage of ?bc solver to obtain a solution; and the third one is to transform the output back to ?ac.

From the first and second concepts we derive the definition of applicable $\mathrm{ARR}^{\Sigma}$,
Definition 5.2.24 (Applicable $\mathbf{A R R}^{\Sigma}$ ) Let be ? arr:d_ARR_Algebraic, such that ? arr [A -> ?ac] and ?arr [B -> ?bc], ?F:d_Formulation, ?s:?ac $\in$ ?F, and ?IW:d_InstanceWrapper, such that ?IW[formulation -> ?F], then ?arr is applicable to reformulate ?s iff

- if ?ac::d_LeafProblem_C and ?ac[not local] then ?IW[global -> ?gdh].
- if ?ac::d_LeafProblem_C and ?ac[local] then ?IW[local(?s) -> ?ldhs].
- if ?ac:: d_Block_C then ?IW[global -> ?gdh] and $\forall$ ? sub[local] $\in$ the substructure tree of ?s ?IW[local(?sub) -> ?ldhs].
- if ?bc::d_LeafProblem_C and ?ac[local ] then ?DH::d_DataHandler, such that d_DHsubscription(?subC, ?DH, ?, ?_) and ?DH:d_Writer.
- if ?bc::d_Block_C then $\forall$ ? subC[local] $\in$ the substructure tree of ?bc, ?DH::d_DataHandler, such that d_DHsubscription(?subC, ?DH, ?, ?_) and ?DH:d_Writer.
- if $\mathrm{bp}_{1}$ if the blueprint for ? ac and $\mathrm{bp}_{2}$ if the blueprint for ? bc then $\mathrm{bp}_{2} \sim \mathrm{bp}_{1}$.

As can be noticed, an applicable $\mathrm{ARR}^{\sum}$ depends on the definition of the proper data handlers and in some cases depends also on blueprint equivalence. After applying the $\operatorname{ARR}^{\Sigma}$, the formulation remains well-formed, because the track structure is innocuous (cf. Definition 5.1.5).

Once the reformulated structure has been substituted by the corresponding track structure, the system needs to handle this kind of structure (e.g. when creating the structured instance). There are two main extensions that must be done in I-DARE(ei) to handle track structures: the first one is related with the IW access methods being able to handle track structures and the second one is related with capacity of the handler of writing a structure (see in Definition 5.2.24 the usage of ?DH::d_DataHandler such that ?DH:d_Writer).

## Extending the IW access methods

In presence of a track structure accessing the data becomes a little bit tricky. We can treat the track structure accessing as a non-global accessing (see §3.1.4) hence using a term like, ?S(?X) where ?S is the structure class containing ? X , and ? X is a dMV or a parameter identifier.

But first of all, what do we need to know about a track structure? Since a track structure ?ts is the product of several transitive $\operatorname{ARR}^{\Sigma}$ applications, it seems logical that we would like to access the data corresponding to tail (?ts) (the last structure in the sequence of reformulations).
 and assuming $S_{2}$ has a constant property $c$ that is computed by an expression $f\left(d_{1}, d_{2}\right)$, where $d_{1}$ and $d_{2}$ are properties of $s_{1}$; then we would like to ask for the value of $\operatorname{tr}\left(s_{1}, S_{2}\right)(c)$. The thing that happens is that, since $c$ depends on $d_{1}$ and $d_{2}$, the system will retrieve their value first, and afterwards it will compute $f$ to obtain the value of $c$.

The process showed in the previous example can be generalized as follows. Let ?ts be a track structure obtained by a transitive sequence of $\mathrm{ARR}^{\sum}$ applications, ? fi $\in$ freeinds(?ts) ${ }^{\xi}$ and ?x be a property (or a dMV) belonging to tail (?ts), and let access be one of the following access methods:
dimSize, constVal; then to compute ?x(? fi ) value, the ? ts(? $\times($ ? fi )) will be used as a parameter of access. The system will then compute the term ? ts(? $\times(? f i))$ in the following way,
$\operatorname{access}(? \mathrm{ts}(? \times(? \mathrm{fi})))= \begin{cases}\operatorname{access}(? \mathrm{ts}(? \times(? \mathrm{fi}))) & , \text { if tail(?ts)}=? \mathrm{C} \text { and } ? \mathrm{ts}: ? \mathrm{C} \\ \mathrm{f}\left(\operatorname{access}\left(\mathrm{rest}(? \mathrm{ts})\left(? \mathrm{~d}_{1}(? \mathrm{fi})\right)\right), \ldots, \operatorname{access}\left(\text { rest }(? \mathrm{ts})\left(? \mathrm{~d}_{m}(? \mathrm{ff})\right)\right)\right) & , \text { otherwise }\end{cases}$
where f and $\mathrm{d}_{1}, \ldots, \mathrm{~d}_{m}$ are extracted from the $\mathrm{ARR}^{\sum}$ between tail (rest (? ts )) and tail (? ts).
Observe that the first case is a non-global access, because ?ts is a leaf problem. Therefore it can be normally handled by the access methods defined in §3.1.4. Also note that we did not mentioned the upperBnd and lowerBnd method, this was because those method at the end will directly depend on constant values that will be accessed by the constVal method.

## The writers

Another important extension to be done to I-DARE(ei) is the inclusion of the data handlers with writing capacity, also called writers.

A writer is a ?DH::d_DataHandler that it is also an instance of the following class,
Listing 5.6: Writer class

```
d_Writer
[
    WdataTag(?IW, ?ts, ?ID) => _list
].
```

where WdataTag(?IW, ?ts) must generate a list of XML nodes to be inserted in the Meta Data file, but it will also generate the actual data, the XML Nodes refer to. The data will be generated depending on the track structure (note that the writer must be subscribed to tail (?ts)).

The writer obtains all the data it needs by accessing the properties of tail (?ts), instancing the free indices using the ?ID dictionary. The implementation of a writer subscribed to a local structure class depends on the format we are trying to use. However, in case of global structure, the writer will always exist, and will depend on an extension of the Meta Data generation algorithm (MDGA) and the d_XMLDataHandler.

When the MDGA encounters a track structure ?ts, it will call for each ? fiv $\in$ ?ts.freeinds ${ }^{\xi}$, the corresponding WdataTag(?IW, ?ts, ?ID), where ? IW is the instance wrapper used by the MDGA and ?ID is the index-value dictionary made from ? ts. freeinds and ? fiv. The MDGA will automatically generate the list of variables corresponding to head(?ts).

The WdataTag(?IW, ?ts, ?ID) of d_XMLDataHandler, will create XML nodes exactly equal to those generated by MDGA, only that in this case they will be generated only based on the information provided by ?ts. Tags <LEAF> are created if tail (?ts ):: d_LeafProblem_C, and the generation is almost the same to the one exposed in $\S 3.2 .1$. Tags <BLOCK> are created if tail (?ts):: d_Block_C. In this case the generation is more complicated, we may use the narrowed block tail (?ts) and create the corresponding block tree, with only the information provided by ?ts. The resulting <BLOCK> is exactly the equal to the one MDGA would have been generated if tail (?ts) had been a block directly created in the I-DARE(im) model.

## Transforming the answers and solvers for track structures

After applying several $\mathrm{ARR}^{\Sigma_{S}}$ to a formulation ?F we obtain a reformulation ?F' that has track structures inside. A track structure (?ts) allowS ?F' to stay well-formed, thanks to the fact that what we are really doing is obtaining a solution of head(?ts) by solving tail (?ts). The input transformation is done a priori by using the adequate writers. Thus, the structured instance will contain the data of tail (?ts). On the other hand, the answer transformation must be done a fortiori by automatically generating $\mathrm{C}++$ methods that maps the operations done in the ans_map (see Figure 5.1).

In general, when the system encounters a track structure ?ts created by the application of a sequence of ARRs, it will generate a set of solvers to deal with ?ts. A solver which structure is a track structure, will be called delegation solver, and it must be an instance of the following class,


Figure 5.1: $\mathrm{ARR}^{\Sigma}$ reformulation and solving process

Listing 5.7: Delegation solver $\mathcal{F}$ LORA-2 class

```
d_delegation_solver :: d_solver
[
    [inner_ds => d_delegation_solver] // optional
].
```

where inner_ds is an optional field, that when present means that there is a dependence on another delegation solver, stored in inner_ds. On the other hand when inner_ds is not present it means that the dependence will be of a normal solver, that will be assigned during the Solvers' Tree generation.

The delegation solver's generation changes depending on how the track structure is formed. Since, until this point, the only way we know to create a track structure is by using a sequence of $A R R^{\Sigma}$, then we will focus now on how the delegation solver is generated for a sequence of ARR ${ }^{\Sigma}$.

Like all other solvers, a delegation solver will have a $\mathcal{F}$ LORA- 2 side and a $\mathrm{C}++$ side. If we are in presence of a track structure ?ts created by a sequence of $A R R^{{ }^{\Sigma}} \mathrm{s}$, then we will generate a delegation solver ds: d_delegation_solver, such that ds[structure -> ?ts] and inner_ds is not present.

Note that ds will directly depend on the solver assigned to tail (?ts). We can "ignore" the structures between head(?ts) and tail (?ts) because all reformulation rules used were ARR ${ }^{\sum}$, therefore, due to the composition of the algebraic operators, we can compose the mappings, and generate the instance for tail (?ts) directly.

This delegation solver ds will have a $\mathrm{C}++$ representation. In $\mathrm{C}++\mathrm{ds}$ will inherit from the solver registered as blueprint of head(?ts), to ensure compatibility with any potential upper solver that was expecting that blueprint. Now, since the blueprint for tail (?ts) (bp_tail) is equivalent to the blueprint for head(?ts) (due to the Definition of $\operatorname{ARR}^{\sum}$ applicability 5.2.24), then ds will define all its methods by calling the corresponding methods in bp_tail.

Assume that the system assigned the solver S to tail (?ts) and that all solvers in $\mathrm{C}++$ have a "sol solve(si)" method, with si being a piece of structure instance, and sol (return type) is equal to the blueprint's retType. Then the ds. solve(si) will call S.solve (si) (si is left intact, since it was already transformed). The only problem is that S.solve(si) will return an answer for tail (?ts) and we need an answer for head(?ts), the answer mappings must then be applied. The answer transformation will be done by automatically creating an auxiliary method in ds,

1 sol_class answer_transformation (sol_class) \{...\}
Let bp be the blueprint of head(?ts), then ? sol_class = bp.retType. The returned object, will be the same
passed as a parameter, just changing the variables' information, by applying the defined ans_map. Note that even if ? sol_class contains information beyond the variable values, this information will remain untouched by the transformation process.

So, ds.solve (si) will call answer_transformation (S.solve(si)), and this way the returned solution will have the correct format. Finally the solver ds is automatically compiled and left ready to be used by the solution process.

The generation of the Solvers' Tree will also be affected by the track structures. Each time the Solvers' Tree generator finds a track structure ?ts, entirely created by the application of ARR ${ }^{\Sigma_{S}}$, it will use the term ?ds(? rs), where ?ds is the delegation solver automatically generated to deal with ?ts and ?rs is the solver that will be used to deal with tail (?ts). Note that ?rs may be a complex term if tail (? ts ): d_Block_C.

### 5.3 Algorithmic ARR

When we apply a sequence of $\operatorname{ARR}^{\Sigma}$ to a formulation, we have two main advantages, the first one is that all the input data is reformulated a priori, so the solution process only has the overload of transforming back the answers. The second advantage is that the application of a transitive sequence of $\mathrm{ARR}^{\sum}$ is at the end reduced to a reformulation between the head and tail of that sequence, due to algebraic expression composition.

However, using just $\operatorname{ARR}^{\Sigma_{S}}$ leaves out a set of reformulations with non-algebraic mappings. For instance, let us consider the following structure class,

```
d_shortPathTree_C :: d_LeafProblem_C
[
    dim_var -> [V],
    args -> [d_constant, // root node
    d_vector(d_constant, [V,V]), // arc matrix
    d_vector(d_var, [V])]
].
```

This class represents the shortest path tree problem on complete graphs of $V$ nodes. In this case the vector of variables will contain the shortest path tree, in the commonly used format: $p[$ root $]=$ root, assuming node root is the origin, and $p[i]=k$, where $k$ is the predecessor node to $i$ in the shortest path tree.

There is a natural LP formulation for the shortest path tree problem, given below. Given a directed graph with $V$ vertexes, we can define the cost (the constant) to be $c_{i j}$, where $i, j \in$ $[0 . . V-1]$; and the arc-usage variable with $x_{i j}$, then an LP formulation for the shortest path tree problem is,

$$
\begin{array}{ll}
\min & \sum_{i j} c_{i j} x_{i j} \\
\quad \sum_{j} x_{i j}-\sum_{j} x_{j i}=\left\{\begin{aligned}
V-1 & \text { if } i=\text { root } \\
-1 & \text { otherwise }
\end{aligned} \quad i \in V\right. \\
\quad x_{i j} \geq 0 & i \in V, j \in V
\end{array}
$$

This LP has the special property that it is integral [18] (i.e., it has an optimal integer solution), returning in each variable $x_{i j}$ the number of paths of the shortest path tree that pass through arc $(i, j)$. So this allow us to easily create the arg_map of an $\mathrm{ARR}^{\Sigma}$ between d_shortPathTree_C and d_LP_C.

Nevertheless, the variables returned by the LP reformulation are not algebraically transformable in the variables that specify the shortest path tree class. Indeed, while the LP has an optimal integer solution, the solvers may not return it. In fact, a solver may return a fractional solution, or even one with oriented cycles, if the graph contains zero-cost cycles.

In fact, the ans_map needed in this example has to be designed as an algorithm that converts the $x$ in the vector representing the path tree. This algorithm will visit, just once, each arc with flow
greater than 0 , to reconstruct the vector p that will be the solution of shortest path tree problem. For instance, if all $\mathrm{p}[\mathrm{i}]=-1$ initially, then an elementary implementation would be

Listing 5.8: Generate the path tree

```
generatePathTree(node, p)
{ // p will be the path tree
    if (node == root) then // just when node is root
        p[node] = root
    for each arc (node, k)
        if (x(node, k)>0 and p[k] == -1) then // flow greater than O (the arc is in use)
                            // and k is not visited yet
            p[k] = node
            generatePathTree(k, p) // starting from k repeat the process
}
```

This example shows that not always mappings can be done algebraically. In fact, in this case an algorithm was necessary to reformulate the answers.

For this kind of reformulations, I-DARE( t$)$ defines the Algorithmic ARRs $\left(\mathrm{ARR}^{\mathcal{A}}\right)$. $\mathrm{An}_{\mathrm{ARR}}{ }^{\mathcal{A}}$ must be an instance of the following class,

Listing 5.9: $\mathrm{ARR}^{\mathcal{A}}$ class definition

```
d_ARR_Algorithmic::d_ARR
[
    indexA => _list,
    dimRel => -list,
    arg_map => _list ,
    fixTI }=>\mathrm{ _list // only if B::d_Block_C
    [condition => -term]
].
```

As can be seen an $A R R^{\mathcal{A}}$ shares some methods with the $\mathrm{ARR}^{\Sigma}$ class definition. The main difference is that all those methods are optional. This optional methods allow us to describe a partially algebraic $A R R^{\mathcal{A}}$, by describing the argument mapping. It is mandatory to also define dimRel and fix TI (in case of blocks) if we are defining arg_map, otherwise the algebraic argument mapping will be completely ignored. For simplicity in the notation we will denote the partially algebraic $A R R^{\mathcal{A}}$, with the symbol, $\mathrm{ARR}^{\mathcal{A}_{\Sigma}}$.

### 5.3.1 Semantics

For $\mathrm{ARR}^{\mathcal{A}_{\mathrm{S}}}$ there is also the concept of applicability,
Definition 5.3.1 (Applicable $\mathbf{A R R}^{\mathcal{A}}$ ) Let be ?arr:d_ARR_Algorithmic, such that ?arr [A -> ?ac] and ?arr [B -> ?bc], ?F:d_Formulation, ?s:?ac $\in$ ?F, and ?IW:d_InstanceWrapper, such that ?IW[formulation -> ?F], then ?arr is applicable to reformulate ?s iff

- if arg_map is present in ?arr then
- if ?ac:: d_LeafProblem_C and ?ac[not local] then ?IW[global -> ?gdh],
- if ?ac::d_LeafProblem_C and ?ac[local] then ?IW[local(?s) -> ?ldhs].
- if ?ac::d_Block_C then ?IW[global $->$ ?gdh] and $\forall$ ?sub[local] $\in$ the substructure tree of ?s ?IW[local(?sub) -> ?ldhs].
- if ?bc::d_LeafProblem_C and ?ac[local] then ?DH::d_DataHandler, such that d_DHsubscription(?subC, ?DH, ?_, ?_) and ?DH:d_Writer.
- if ?bc::d_Block_C then $\forall$ ?subC[local] $\in$ the substructure tree of ?bc, ?DH::d_DataHandler, such that d_DHsubscription(?subC, ?DH, ?_, ?_) and ?DH:d_Writer.
- if $\mathrm{bp}_{1}$ is the blueprint of ?ac and $b p_{2}$ is the blueprint of ? bc then $\mathrm{bp}_{2} \sim \mathrm{bp}_{1}$.
- There must be a delegation solver ds, such that ds. structure = $\operatorname{tr}($ ? ac, ? $b \mathrm{bc})$, explicitly defined (not automatically).
Note that if ?arr is an $\mathrm{ARR}^{\mathcal{A}_{\Sigma}}$ then the delegation solver ds must only implement the answer mapping, like it was done for $\mathrm{ARR}^{\sum}$, however, ds have to be explicitly defined. On the other hand, a ds used for a $A R R^{\mathcal{A}}$, will transform (a fortiori) the structure instance to be passed as argument to the solver for ?bc and transform back the solution. See that all these transformations are hard-coded in the $\mathrm{C}++$ source of the ds.

The main difference between $\operatorname{ARR}^{\mathcal{A}}$ and $\operatorname{ARR}^{\Sigma}$ is that $\mathrm{ARR}^{\mathcal{A}}$ depends on the existence of delegation solvers and $\mathrm{ARR}^{\Sigma}$ does not. However when the system encounter a track structure (?ts) entirely created by the application of $\mathrm{ARR}^{\mathcal{A}} \mathrm{s}$, it will automatically generate a dummy delegation solver (dummy_ds), such that dummy_ds[structure -> ?ts]. This dummy_ds will use the inner_ds field to define which delegation solver it will depend on. The dummy delegation solver is just used in the $\mathcal{F}$ LORA-2 side (mainly by the Solvers' Tree generator). Let us describe, how a delegation solver will be assigned to dummy_ds.inner_ds.

Assume, ?ts is formed by the application of $K \operatorname{ARR}^{\mathcal{A}} \mathrm{s}$ (and/or $\mathrm{ARR}^{\mathcal{A}_{\Sigma}}$ ), and that ds $\mathrm{s}_{j}$ is the already existing delegation solver, such that $\mathrm{ds}_{j}$. $\operatorname{structure~}=\quad \operatorname{tr}(? \mathrm{ts}[j-1]$, ? ts $[j])$, with $j \in[1 . . K]$, then dummy_ds.inner_ds $=d s_{1}, \mathrm{ds}_{j}$.inner_ds $=\mathrm{ds}_{j+1}$, for all $j \in[1 . . K-1]$. Observe that the dummy_ds defines a sequence of delegation solver concatenated calls, that will eventually produce the call of the solver assigned to tail (?ts). Also note that all the delegation solvers used to deal with ?ts have a blueprint equivalent to the blueprint of head(?ts).

This previous dummy solver generation only works if the sequence is purely of $\mathrm{ARR}^{\mathcal{A}}$ (and/or $\left.A R R^{\mathcal{A}} \Sigma\right)$. But, what happens if the sequence also contains $\operatorname{ARR}^{\Sigma_{s}}$ ? The treatment when we have a mixed sequence will depend on how it is formed. For instance, it may have an initial sequence of $A R R^{\Sigma}$, followed by a mixed sequence. The following list will describe the two possible structure lists that may form a track structure,
$\mathbf{L S 1}=\mathrm{b}_{0}, \ldots, \mathrm{~b}_{m_{L S 1}}$ such that from 0 to $m_{L S 1}$ was created by $\mathrm{ARR}^{\sum_{\mathrm{S}}}$;
$\mathbf{L S 2}=\mathrm{b}_{0}, \mathrm{~b} 1, \ldots, \mathrm{~b}_{m_{L S 2}}$, such that from 0 to 1 was created by an $\mathrm{ARR}^{\mathcal{A}}$ (or an $\mathrm{ARR}^{\mathcal{A}_{\Sigma}}$ ), and from

Using LS1 and LS2, we will describe how the dummy solver will be created for the different possible track structures. For doing so we will use a function called assembler, defined as follows.
Definition 5.3.2 (Assembler function) Given the structure lists LS1 and LS2, then the assembler function of a track structure ?ts is defined as follows, assembler(? ts ) $=D$, such that $D$ : d_delegation_solver and

- if ? ts $={ }_{\_} \operatorname{tr}(\mathrm{LS} 1)$ then $D$ is automatically generated, $D$. structure $=$ ?ts and no inner_ds is defined in $D$.

- if ? ts $=\operatorname{tr}^{\operatorname{tr}(\mathrm{LS} 1, \mathrm{LS} 2)}$ then $D$. structure $=\operatorname{\_ }^{\operatorname{tr}(\mathrm{LS} 1)}$ and $D$. inner_ds $=$ assember(_tr(LS2)).

Finally the dummy solver of a mixed track structure ?ts will be dummy_ds:d_delegation_solver, such that dummy_ds.structure $=$ ?ts and dummy_ds.inner_solver $=$ assembler(?ts).

Even if the system recognizes the presence of sequences of $\mathrm{ARR}^{\Sigma}$ applications embedded in a mixed track structure, and it automatically generates the corresponding delegation solver, we can not always do a priori structure instance mapping. In fact if the sequences of $\mathrm{ARR}^{\Sigma}$ is preceded by at least one $\operatorname{ARR}^{\mathcal{A}}$ then no a priori structure instance mapping ca be done. The automatically generated solver must implement a method to map the structure instance like the one to map back the solution. Such a method must be called before calling the inner solver. Notice that when we are in presence of a sequence (or initial subsequence) of only $A R R^{\Sigma_{S}}$ and $A R R^{\mathcal{A}_{\Sigma}}{ }_{\mathrm{S}}$, the system, although not using just one solver (due to the $A R R^{\mathcal{A}_{\Sigma}}$ ), it can make a priori structure instance mapping (due to the arg_map present in the $\operatorname{ARR}^{\mathcal{A}^{\mathcal{A}}} \mathrm{s}$ ).

The Solvers' Tree of a mixed track structure will be computed using the same methodology exposed in the $\mathrm{ARR}^{\Sigma}$ 's case, but using the dummy delegation solver instead.

### 5.4 Selection domain and Reformulation Domain

I-DARE( t ) will apply the defined ARRs to different parts of a formulation. It may decide to reformulate a leaf problem, a whole block, or even a subcomponent inside of a narrowing. All the potentially transformable components in a formulation will belong to a set called selection domain.

Definition 5.4.1 (Selection domain) Let F be a formulation, then the selection domain of F , denoted by $\Omega(\mathrm{F})$, is defined as follows,

- $\operatorname{F}$.root $\in \Omega(\mathrm{F})$,
- if b:d_Block_C $\in \Omega(\mathrm{F})$, then $\forall($ sub $\in$ b.subs $)[$ sub $\in \Omega(\mathrm{F})]$,
- if ts $=\operatorname{tr}(\ldots) \in \Omega(\mathrm{F})$ and tail ( ts ) is the narrowing $\mathrm{b}\left(\mathrm{s}_{1}, \ldots, \mathrm{~s}_{k}\right)$, then the term $<\mathrm{s}_{i}$, $\mathrm{b} . \mathrm{ids}[i]$, ts $>\in$ $\Omega(\mathrm{F}), \forall i \in[1 . . k]$,
- if $<\mathrm{s}$, id, path $>\in \Omega(\mathrm{F})$ and s is the narrowing $\mathrm{b}\left(\mathrm{s}_{1}, \ldots, \mathrm{~s}_{k}\right)$, then the term $<\mathrm{s}_{i}$, b.ids $[i],<\mathrm{s}$, id, path $\gg \in$ $\Omega(\mathrm{F}), \forall i \in[1 . . k]$.

Note that since all structures in $F$ have a different name (see Definition 2.6.1), the terms created to access each narrowing will also be unique.

For instance assume we have a formulation F represented in Figure 5.2 then selection domain


Figure 5.2: Formulation example
of F will be,

$$
\Omega(\mathrm{F})=\{\mathrm{a}, \mathrm{~b}, \mathrm{c}, \mathrm{~d}, \operatorname{tr}(\mathrm{e}, \mathrm{G}(\mathrm{H} 1, \mathrm{H} 2)),<\mathrm{H} 1, \mathrm{id} 1, \quad \operatorname{tr}(\mathrm{e}, \mathrm{G}(\mathrm{H} 1, \mathrm{H} 2))>,<\mathrm{H} 2, \mathrm{id} 2, \quad \operatorname{tr}(\mathrm{e}, \mathrm{G}(\mathrm{H} 1, \mathrm{H} 2))>\}
$$

Using the track structures we can transitively apply ARRs. For instance, if we have a structure ?s:? cs and an ARR ? arr, such that ? arr. A = ?cs, then applying ?arr we will obtain the track structure _tr (?s, ? arr.B). But, if we start with _tr (?s, ? arr.B), and have an ARR ? arr ${ }_{1}$, such that ? arr $1 . A=$ ?arr.B, then we will obtain the track structure _tr (?s, ? arr.B, ? arr ${ }_{1} . \mathrm{B}$ ), and so on.

This transitive ARR application, will conduct to the generation of a set formulation that will be deduced from the original one. The process of applying an ARR to a structure in a formulation F to obtain a new formulation, will be called $A R R$ usage function.

Definition 5.4.2 (ARR usage function) Given a formulation F:d_Formulation, a structure $\mathrm{s} \in \Omega(\mathrm{F})$ and an arr:d_ARR such that arr is applicable to s , then the $A R R$ usage function of arr to reformulate $s$ in $F$, denoted by $\kappa(\mathrm{F}, \mathrm{s}$, arr $)$, produces a new formulation by substituting $s$ in F with a track structure created from applying arr.

If we start from an initial formulation $F$, we can apply $\kappa(\mathrm{F}, \mathrm{s}$, arr $)$ to obtain new formulations, by changing s and arr, and even $F$. This sequenced application of the ARR usage function will produce a set containing all possible reformulation starting from $F$, this set will be called $\Xi(F)$.

Proposition 5.4.3 Given a formulation F:d_Formulation, then $\Xi(F)$ is finite.

Proof. Since the amount of ARRs is finite, the amount of structures inside F is finite and the track structures can not contain cycles (see Definition 5.1.4), so we will arrive to a point where no new ARR can be applied, because it would generate a cycle in the track structure, hence no new formulation will be derived. Therefore $\Xi(F)$ is finite.

### 5.5 Discussion

The I-DARE $(\mathrm{t})$ is an inference machine that, based on a set of ARRs, transforms one well-formed formulation into another (also well-formed). Since it requires querying the model and data, it is also implemented using declarative programming ( $\mathcal{F L O R A}-2$ ). In fact the implementation, if done in $\mathcal{F}$ LORA-2, will be almost equal to the definition of the system, except for the mappings that sometimes may be non-declarative by nature. In those cases we use the interoperability between $\mathcal{F}$ LORA-2 and other languages like $\mathrm{C}++$, to implement certain procedural processes.

In contrast with our definition of reformulation, the concept used in [143] does not offer algorithmic notions; therefore, its implementation may be rather difficult. Furthermore, it leaves a huge amount of possible reformulations out because of the restricting conditions it imposes. On the other hand, [24] manages the idea of mapping functions; from that point of view, it has the same power that our reformulation system has. However, we propose a reformulation system defined over a precise modeling language, that allows us to algorithmically and algebraically deduce reformulations. I-DARE $(\mathrm{t})$ offers a way of determining which structures can be reformulated and how they will be reformulated, obtaining at the end of the process valid formulations and data ready to be given to the solvers.

The study in $[113,114]$ define a more complete framework for making automatic reformulations, via symbolic transformations applied to the sets of variables, objectives and constraints. They formally define a set of reformulation techniques that covers a good number of real-life problems. However, they are based on algebraic transformations of formulations that do not make explicit use of the structures, and therefore may not be able to exploit the semantic value of certain structures. This fact provokes the exclusion of reformulations that treated in a structured way may be possible to achieve algorithmically.

I-DARE ( t ) offers a methodology based on deduction that allows us to deal with complex and non-direct reformulations. This methodology is fully based on the fact that our model is structured and that we know how to deal with these structures (in a solution process or at least when defining the ARRs). We may construct a structured formulation for which we do not have a solution process, but we may obtain a reformulation, applying I-DARE ( t$)$, for which all its structures have a solver "attached".

One of the main gains by using $\operatorname{I-DARE}(\mathrm{t})$ is the extensibility. For instance, we can widen the amount of reformulation rules, using the techniques in [113], doing so we can cover several algebraic reformulations. Furthermore, I-DARE ( t ) offers the possibility of selecting intelligently between possible reformulations, by changing the parameters of the $\kappa$ function.

Moreover, there is a very useful application that derives naturally from the creation of $\mathrm{ARR}^{\Sigma}$, and it is a format transformation mechanism. This format transformation is based on the handlers definitions and the creation of pretty simple $\operatorname{ARR}^{\Sigma_{S}}$. For instance, we could easily create an ARR ${ }^{\Sigma}$ to reformulate d_LP_C into d_LP_MPS_C, once the the data handlers for d_LP_MPS_C are adequately specified. Therefore I-DARE( t$)$ may become a huge format transformation engine, as it grows in structure classes, ARRs and data handlers.

## Chapter 6

## I-DARE(control) - best (Formulation, Solver, Configuration)


#### Abstract

The main aim of I-DARE is to produce models that can be automatically and algorithmically reformulated to search for the "best" formulation, intended as the one for which the most efficient solution approach is available. This requires exploration of a high-dimensional space comprising all (structured) reformulations of a given instance, all available solvers for (each part of) the formulation, and all possible configurations of the relevant algorithmic parameters for each solver. A fundamental pre-requisite for this exploration is the ability to predict the efficiency of a given (set of) algorithm(s), considering their configuration(s), for a given instance; this is, however, a vastly nontrivial task. This chapter describes how the information is organized to make the search in the (formulation, solver, configuration) space possible with several different exploration techniques. In particular, we propose a way to combine general machine learning (ML) mechanisms and ad-hoc methods, where available, in order to effectively compute the "objective function" of the search. We also discuss how this mechanism can take upon itself part of the exploration, the one in the sub-space of configurations, thus simplifying the task to the rest of the system by reducing the dimensionality of the search space it has to traverse. Finally we present some practical results using MCF structure and SVR ML technique.


### 6.1 Search Spaces

I-DARE (via the queries provided by $\mathcal{F}$ LORA-2 [159]) retrieves structured data about the current instance, in a very effective way. This data can be used to characterize and explore the search space, that is composed by three main sub-spaces:

- Formulation + Instance $=$ Extended Model;
- Solvers;
- Configurations.

For each sub-space, an extensible set of predefined queries and methods are available to consult the data. These queries provide any control mechanisms (cf. §6.2) with the information it needs for effectively guiding the search throughout the whole space.

### 6.1.1 Extended Model

A large number of queries are available in I-DARE to retrieve information about variables, constants, dimensions, and components of an Extended Model (EM); how these components are related between each other within a formulation. These queries allow to obtain a complete description of any "static" EM. To obtain this description one may use the methods present in I-DARE(im) that allow to access the formulation information, for instance,

- names of dimensions, indices, constants and variables;
- class of a structure;
- information about the structure tree;
- how variables are shared throughout the formulation.

Moreover, once created the instance wrapper over that formulation, one can access the information related with the actual data, for example,

- cardinality of dimensions and indices;
- values of constants;
- bound values of variables.

Since the queries set is extensible, new queries can be implemented to support all kind of moves in the formulations space, such as breeding in population-based heuristics, basin-hopping in local search, and many others.

However, the main goal of I-DARE is to allow reformulating the models. This is obtained by applying atomic reformulation rules (ARR) (see §5.1) to specific components inside the formulation. As can be seen in $\S 5.4$, the ARRs are applied using the function $\kappa(\mathrm{F}, \mathrm{s}$, arr $)$, where F is a formulation, $s$ is a structure in $F$, and arr is an atomic reformulation rule applicable to s. Note that $\kappa$ defines the search space of all reformulations of F. Furthermore, it defines the search space of all EMs that derive from F by applying Algebraic ARRs, because we can map the instance data a priori.

### 6.1.2 Solvers and Configurations

I-DARE defines a general interface for solver plug-ins. Each solver must register itself to one structure and define its configuration template. I-DARE automatically generates a $\mathcal{F}$ LORA- 2 file containing solver and configuration data. Each structure class in I-DARE(lib) may have more than one solver registered. This database defines the solver's search sub-space, which can be consulted using the defined instances of d_solver to retrieve registration and/or configuration information.

As previously mentioned, each solver must define how it must be configured. For this purpose I-DARE defines Configuration Templates (CT). A CT is a hierarchical structure defining the relevant algorithmic parameters and the possible range of their values. Hence, CTs can be easily used to describe single configurations by simply forcing each parameter to have a single-valued domain.

Two descriptions of CTs are available: the "external" and the "internal" one. The external one is in terms of an XML file that specifies parameters and their domains. CTs currently support four base parameter types: integer, double, choice and vector. When a solver is exported to the $\mathcal{F}$ LORA-2 file, it exports also its CT. Therefore, the "internal" representation of the CT in $\mathcal{F}$ LORA-2 is automatically constructed (cf. §4.4.2).

### 6.2 Controlling the Search in the (Formulation, Solver, Configuration) Space

All the I-DARE components described so far are conceived for providing the basic blocks for the most delicate and innovative feature of the system (I-DARE(control)): given a structured instance,
to automatically select the "best" combination in the space of the possible (re)formulations, solvers and configurations. That is, one must select one particular (re)formulation among all the possible ones obtainable by the atomic reformulation rules, select an appropriate solver-among the possibly several available ones-for each node in the formulation tree, and select an appropriate configuration-among the possibly very many choices - for each of the solvers.

This is clearly a very complex process, for which several different techniques may be used. In principle, of course, it requires the solution of an appropriate "meta" optimization problem in a suitably defined space. However, the problem is made particularly difficult by the fact that even predicting the performances of a given (set of) algorithm(s) and configuration(s) on a given formulation and instance is far from being a trivial task.

We have chosen to provide a rather general and abstract setting for performing the search, so as to allow different search mechanisms to be compared and contrasted. The whole search is controlled by the I-DARE (control) module, which may be any control mechanism conforming to the simple interface

```
d_control [
    process(d_InstanceWrapper) -> [d_InstanceWrapper, _term]
] .
```

This interface declares a method that, given an extended model, returns the selected "best" reformulation of the model along with the solver tree and the correspondent configuration. Of course the initial and final model may be the same, in which case I-DARE(control) "only" selects the best solver and configuration for the given instance. This is already a rather difficult problem in itself, for which little is known in practice; indeed, it is important to remark that even predicting the running time of a given algorithmic approach on a given data input is problematic. While there is a huge literature about the theoretical complexity and practical performances of the countless many different algorithms for each of the many possible structures the I-DARE aims at eventually capture, very little is available in terms of methods capable of taking this kind of decision in a general setting.

This seems to essentially require the use of Machine Learning (ML) techniques (e.g. [38]), which may be the only approach capable of automatically devising suitable approximations of the function which estimates the efficiency (and, possibly, the effectiveness) of an algorithmic approach when applied to the solution of a given instance. As we shall see, the use of ML tools, besides being necessary to evaluate the "objective function" of the search, provides a natural way for actually performing a part of the search, in particular that in the subspace of algorithms and configurations. Remarkably, the use of ML tools for the selection of algorithm parameters have been recently advocated in [49], although in a much more limited context, with promising initial results.

Thus, while the I-DARE system does not specify the exact strategy used by the I-DARE(control) module to search the (Formulation, Solver, Configuration) space, it must provide any actual implementation with enough information to effectively drive the process. While I-DARE(control) has full access to all the characteristics of the instance (cf. §6.1.1), the previous discussion highlights the need for further mechanisms that allow an efficient comparison between different points of the space. These are described in the next sections.

### 6.2.1 Objective function computation

The fundamental mechanism needed for driving the search is an effective and efficient way for evaluating the quality of a (Formulation, Solvers, Configuration) choice; we will consider this the "objective function" of the search, and denote it by $\psi$. At first reading, one may imagine that $\psi$ measures the running time required by the solver, with the specified configuration, to solve the corresponding instance; however, different cases are also possible.

For instance, since many problems are "hard", it may well be impossible to solve them to proven optimality in a reasonable amount of time. In this case, the user would typically set a desired target accuracy, and a maximum time limit. Hence, $\psi$ should now account for the running
time it takes to the solver obtain a solution with the prescribed optimality, if that can be done within the time limit, and a weighted sum of time limit and final objective function gap otherwise. In this way, the fastest solver capable of attaining the desired accuracy within the limit is selected, if there is any, and the solver providing the most accurate solution at the end of the allotted time is selected otherwise. Alternatively, accuracy of the solution may be treated as a parameter (cf. "Fixed Features" below).

In general, one should not expect that an arithmetic or algorithmic description of $\psi$ be available for all possible formulations, solvers and configurations, although this may indeed happen in some cases. Therefore, we propose the application of ML techniques to approximate such function based on known observations.

## Features

As usual in ML, one critical point is the definition of the set of features that represent each data point in the learning set of the method. It is well-known that the complexity (and practical performances) of several optimization algorithms can be shown to depend in somewhat predictable ways from some well-understood characteristics of the instances: for Linear Programs, for instance, some of the main features are the number of variables and constraints together with the density of the constraint matrix. However, the relevant set of features should be expected to be very different for different problem classes, and even for different algorithms for the same problem class; again in the LP case, degeneracy of the vertices of the polyhedron (that can usually be estimated by some properties of the RHS of the constraints) strongly affects simplex approaches but is next to irrelevant for interior-point ones. Therefore, defining a unique set of features for a problem does not seem reasonable: each solver should be able to specify a different set of features. On the other hand, the responsibility of defining the right set of features cannot be demanded to a general mechanism, so each solver will be required to define them.

Thus, we define a layer over the existing I-DARE solver interface, which is called Solver Wrapper (SW), that will provide the list of relevant features to parametrize $\psi$, i.e., a dictionary [name=val, ....], where name is the nominal representing the feature and val is the value this feature takes. Of course, a SW must ensure that its feature list always contains the same set of names. All SWs must inherit from the following interface

Listing 6.1: Solver Wrapper Interface

```
d_solverWrapper [
    solver }\quad=>\mathrm{ d_solver,
    retrieve(?EM, ?CT) => [_list, CT],
    [internal]
].
```

Given the current EM, the retrieve () method returns the feature list and a list of possible configurations, represented by a CT. The meaning of the method is somewhat different according to the value of the optional property internal.

- When internal is not present, the evaluation of $\psi$ is demanded to the general mechanism described later on. In this case, the SW "only" has the responsibility to extract from EM, that is of course of known type, the features set. The second return value in retrieve () is a CT that is intended to describe all possible configurations (compatible with the fixed choices, see below) of the solver for this particular instance type.
- When internal is present instead, the evaluation of $\psi$ for the given solver is done inside the wrapper. In this case, there will be only one (or few) features, consisting in the (estimated) value(s) of $\psi$ (or, maybe, in the description of the function relating running time with accuracy) for the instance EM. Actually, the very concept of $\psi$ requires that of a configuration attached to the solver, since this choice impacts on the performances. In fact, in this case second return value in retrieve () is meant to contain the single configuration that produces
the estimated value of $\psi$. It is intended in this case that the $S W$ will choose the (estimated) best configuration, if more than one is available.

Actually, since the SW can implement retrieve () in any (sensible) way, there may be intermediate scenarios between these two extreme ones. For instance, the SW may internally compute some sophisticated performance figures, out of which predicting the actual running time may be much easier, and/or return a configuration template containing only a subset of the possible configurations, discarding those that are estimated to be unlikely to prove efficient. This general mechanism allows on one side to use a general ML mechanism (described below) for the case where nothing relevant is known about predicting the performances of a solver, and on the other side to exploit specialized techniques when they are available. Note that the SW may well use, internally, a specialized ML approach to select the best configuration, should one be available (cf. e.g. [49]). Moreover these performance figures may be designed to reflect, for instance, the trade-off between time and solution accuracy (cf. Fixed Features).

## The Generic Machine Learning Sub-system

When a SW does not compute its $\psi$ value internally, the Generic Machine Learning Sub-system (GMLS) can be invoked to try to estimate it.

In general, a SW will produce a features list and a CT. The feature list is dependent only on the specific instance EM, and not on the configuration, whereas the CT is independent from EM. Therefore, this information actually corresponds to several values of $\psi$, one for each configuration in the template (although there may be only one, e.g. when internal is present). In ML parlance, the SW (implicitly) produces several data points, each one formed by the unique feature set of EM and one among the different configurations from the template; in other words, the actual features set of the ML is a pair (features of the instance, configuration of the algorithm).

This information can be used with any of the several possible ML approaches to try to estimate $\psi$; clearly, different approaches may turn out to be more effective for different algorithms. In order not to tie-in the I-DARE system to any specific ML technology, I-DARE(control) defines a general interface to ML algorithms, described by the following class

Listing 6.2: Machine Learning Interface

```
d_machineLearning [
    evaluate(_list) => _list,
    = train(_list, _list)
].
```

where

- train () trains the ML using a set of data points-that is, (features, configuration) pairs-and a list of known $\psi$-values (one for each point);
- evaluate() computes $\psi$ for the specified data point.

Each concrete class inheriting from d_machineLearning will define an actual ML technique (Neural Networks, Support Vector Machine, Decision Tree, ...); the GMLS sub-system will associate each SW with one (possibly the "most appropriate", cf. $\S 6.2 .2$ ) concrete ML in charge of computing $\psi$ for the corresponding solver.

Note that for nested structures (formulations that contain other structured problems as subblocks, cf. §2.5), the SW has the possibility to access the SWs of the sub-blocks and therefore it can (but it does not necessarily need to) exploit their computation of the $\psi$ values for the sub-blocks as inputs for its own computation (either with ML techniques, or with any other mean) of the $\psi$ values for the entire block. This allows to nicely decompose the (difficult) task of prediction $\psi$ for a complex algorithm into the (hopefully, easier) tasks of predicting $\psi$ for each component and then predicting how the individual performances affect the global one.

## Machine Learning as a Search Mechanism

Clearly, the above ML approach provides one way to automatize the search in the configuration space. Provided that the configurations are "few", one may simply list them all and compute $\psi$ for each; then, the configuration with the best value is retained as the selected one. Provided that the possible solvers for a given structure are not too many either (which looks a reasonable assumption), an effective ML approach to computing $\psi$ would provide all the tools for performing the search in the (Solver, Configuration) sub-space, leaving "only" the (re-)formulations space to be explored.

In general, however, the set of configurations may be rather large. One might thus devise ML approaches capable of working with "meta" data points, i.e., pairs (features of the instance, configurations template). These approaches might for instance still rely on standard ML techniques at their core, but coupled with smart sampling techniques that avoid to compute all possible data points, somewhat in the spirit of active learning techniques [141]. More in general, one may devise ML approaches aimed not just at predicting $\psi$ for a given configuration, but rather at predicting the configuration which produces the best value of $\psi$ within a given CT. Some very preliminary steps along this line have already been done e.g. in [49].

## Fixed Features

The retrieve () method of the SW has a second parameter ?CT (that may conceivably be empty), whose use has not been discussed so far. That is intended to be a partial CT, whose use is to constraining the possible configurations to be generated by SW. This allows the caller of a SW to instruct it (in particular, in the case where $\psi$ is computed internally) to avoid considering some configurations that are not feasible, or not "interesting".

There are at least three important cases that may require such a mechanism:

- handling of accuracy in the solution, in terms of either constraint satisfaction or of quality of the obtained solution;
- handling of maximum resource usage (typically, CPU time) in the solver;
- handling of the architecture, i.e., the fact that the same solver may be executed on different parallel hardware (say with a different number of cores, and/or with the presence of specialized hardware such as GPU accelerators).

These aspects may reasonably be considered included in the configuration of a solver. However, depending on the actual form of $\psi$, they may not be freely chosen by the SW in quest for the smallest $\psi$ value. In fact, accuracy of the overall solution and/or the maximum total allotted running time will typically be set by the final user depending on her needs. In turn, a block of the formulation that has some sub-blocks may want to explore their accuracy/time frontier to seek for the most appropriate setting, e.g. settling to (slightly) less accurate solutions in change for a (consistently) reduced running time; this is, for instance, the setting that is most often chosen for separation algorithms in Mixed-Integer Programs when-as it often happens-they require the solution of a hard subproblem.

However, in other cases a "master" problem may require solutions of its subproblems with a higher degree of accuracy from the one of the solutions it is expected to provide. For maximum resource usage, it is clear that, in most cases, subproblems of a more complex formulation will have to be solved in much less time than the maximum one allotted for the whole problem. Finally, a SW may want to explore the possibility to allocate its subproblems to different computational nodes to exploit their complementary strengths (see e.g. [45] for one example); on the other hand, some solvers may not be available (or be known to scale very badly) on some architectures, or the target architecture may be severely limited by the user due to price or availability concerns.

All this cases can be handled with the general mechanism of externally constraining the set of available configurations. Note that if the SW is not able to generate at least one configuration
that satisfies the constraints imposed by ?CT, it will fail by returning an empty CT, thus signaling that it cannot be used under that set of conditions; basically, this amounts at producing an infinite value of $\psi$. This way, inner solvers may "constrain" their outer solvers to avoid some specific configuration parameters.

Note that I-DARE does not, in general, enforces that the parameters set by the partial CT in retrieve () be meaningful for the SW. For instance, some solvers may only be capable of providing exact solutions to their problem, and therefore the accuracy setting may not be meaningful for them. Also, parameters are always dealt with at the syntactic level, and therefore some discipline will be needed in the construction of the I-DARE library to ensure that at least some main parameters (e.g. accuracy, running time and architecture) be uniformly recognized by all solver. Note that, however, checks can be easily put in place so as to ensure syntactic compatibility between CTs, so that at least warnings can be ensued.

### 6.2.2 Training and Meta-Learning

## Training

The fundamental assumption under any ML approach is that the machine be fed with an appropriate set of samples, i.e., data points with the associated value(s) of the function(s) to be learn. This is known as training. The GMLS sub-system will therefore have to execute a learning process before that the ML be ready for actual use in the search. The learning process consists in solving the instances in the training database with all available algorithms and all available configurations, thereby producing the data to be fed to the train method of d_machineLearning. This process can clearly be very time-consuming, and it will have to be (partially) repeated each time either new instances are added to the training database, or solvers are updated/added. Luckily, the learning process can be easily deployed in a parallel environment to take advantage of its high level of inherent parallelism.

## Meta Learning

It is obvious that the effectiveness of the prediction of $\psi$, upon which all the search process ultimately rely, can be very significantly affected by the choice of the concrete ML in charge of computing $\psi$ for any specific SW, together with its possible several learning parameters (topology of the Neural Network, parameters of the Support Vector Machine, ...) [38]. Choosing the "most appropriate" ML is therefore, itself, a difficult (yet fundamental) task. Thus, GMLS will also have to implement a meta learning process, whereby the results of the same learning phase for a given solver are fed into different ML, and the "best" machine is selected as the one which minimizes some appropriate discrepancy measure between the actual values and the predictions. This can be done with the usual procedures, akin to $k$-fold validation, whereby the set of available data is (randomly, in several different ways) subdivided into a training set, that is actually fed to the ML, and a testing set upon which predictions of the ML are computed and contrasted with the (known) true results.

Since all ML share the same interface, this process can be automatized and regularly repeated e.g. whenever the testing database significantly changes; again, while very time-consuming the process is also inherently very parallel. Furthermore, the computationally heavy part-actually executing the solvers on the given instances for the selected configurations-need to be done only once; provided that the results are properly stored, they can re-used by all MLs, and over and over again during subsequent meta-learning phases.

### 6.2.3 The overall search process

The GMLS sub-system thus defined provides a sound basis for implementing any general search procedure in the (formulation, solver, configuration) space; actually, it may also directly take care of the selection of the latter two components (solver and configuration), leaving to I-DARE(control) "only" the task of appropriately traversing the (re)formulations space using the available ARRs to
reformulate parts of the whole structured model. Ultimately, I-DARE(control) has the responsibility of providing the end user with the ((re)formulation, solver, configuration) that is going to be used to actually solve her problem within the allotted time, accuracy and/or monetary budget constraints. This of course requires implementing a search over the formulation space, for which several different approaches are possible, from complete enumeration to (more likely) heuristic searches such as any variant of local search (with taboo or simulated annealing) or population-based searches such as genetic algorithms. It is also possible to apply ML as a search tool, analogously to what it is done for configurations. Figure 6.1 shows a diagram that outlines the overall search process in I-DARE(control), highlighting the fundamental role of GMLS.

It worth mentioning, that each time a final EM is selected and actually solved, all solution data is sent back to I-DARE so as to be added to the testing database. This way, the testing database is automatically enriched from real problems, strengthening the observation set and therefore allowing the GMLS to perform a better approximation of $\psi$ in the future. This may be the source of a positive feedback loop, whereby good performances of the system attract more users, who provide more data which in turn ultimately leads to even increased performances.


Figure 6.1: GMLS diagram

### 6.3 Experiments

In order to test the soundness of the design decisions of the I-DARE system, we performed some preliminary experiments with a relatively simple (yet quite powerful) model: the Min-Cost Flow (MCF) problem. For this we selected two solvers: the primal/dual RelaxIV [36] and MCFSimplex, a recent implementation of the classical network simplex algorithm [19], both distributed by the MCFClass project [124]. Besides being rather different in nature, the solvers have several algorithmic parameters which impact their performances. For MCFSimplex these are the choice of primal or dual simplex, the number of candidate list and the size of hot list in the all-important pricing rule. For RelaxIV one can decide if the auction/shortest paths initialization procedure is used, the number of single-node iterations attempted before the first multinode iteration is allowed, the threshold parameters to stop the scanning process after a multinode price changes, the bound to decide when another multinode iteration must be performed, and the number of passes. Thus, a large
number of possible different configurations can be used besides the default one (Dconf), which is typically hard-coded in the solver and however not touched by all but the most adventurous users.

These solver were applied over 144 different graphs using different configurations. The graphs were created using the following generators: complete networks (10-3000 nodes/106-11252189 arcs), gridgen (255-65535/2048-1048576) and netgen (256-16384/2048-1048600). After having executed each configuration of each solver on each instance we tested the effectiveness of the ML approach as the basis for the search in the (Solver, Configuration) sub-space. For this, a $k$-fold validation mechanism was applied. We partitioned randomly the set of graphs into 4 equally sized chunks. The ML was trained using 3 chunks and tested for accuracy using one, this process was repeated four times ensuring all chunks were part of the testing process (the average results were reported on all trials). Given that MCF is a polynomially solvable problem, the $\psi$ function only measured the obtained running time.

In order to show that a ML approach makes sense, we first report some data showing that choosing the best solver and configuration indeed makes a difference. In the following table we report for each solver the number of different configurations that were found to be the best one (denoted by Aconf) for at least one instance ("nbest"), the average ratio between the running time of Aconf and that of Dconf ("b/d ratio", in parenthesis the variance), and the percentage of total instances in which the best configuration for that solver was better than the best configuration for the other solver ("\% best").

| Solver | nbest | b/d ratio | \% best |
| :---: | :---: | :---: | :---: |
| RelaxIV | 80 | $0.77(0.22)$ | 70.8 |
| MCFSimplex | 34 | $0.86(0.17)$ | 29.2 |

We also mention that the best solver requires on average around $60 \%$ of the running time of the worst solver.

We then experimented about the effectiveness of the ML approach for the selection of the best (Solver, Configuration) pair using the support vector regression (SVR) tools provided by the SHOGUN library [145]. For our tests we analyzed different feature combinations, taken from a set of 24 ones: nodes (0), arcs (1), min/max/average/variance of node degree ( $2,3,4,5$ ), arc capacities $(6,7,8,9)$, arc costs $(10,11,12,13)$, node deficits $(14,15,16,17)$, and length of min-hop path $(19,20,21,22)$, the ratio of average node deficit and arc capacity (18), and an approximation of the graph diameter (23).

The results are shown in the following table; in particular, "e. err" is the average displacement of the estimated time for the best configuration according to ML (MLconf) with respect to the actual time for that configuration, "b/ml ratio" is the ratio between the running time of Aconf and that of MLconf, "ml/d ratio" is the ratio between the running time of MLconf and that of Dconf, and "opt. features" is the subset of features that have been found to provide the most accurate results.

| Solver | e. err | b/ml ratio | ml/d ratio | opt. features |
| :---: | :---: | :---: | :---: | :--- |
| RelaxIV | $0.542(0.871)$ | $0.770(0.205)$ | $0.966(0.305)$ | $[0,1,4,8,12,16,24]$ |
| MCFSimplex | $0.500(0.723)$ | $0.855(0.182)$ | $0.919(0.158)$ | $[0,1,18,21,23]$ |

We emphasize that different feature subsets led to fairly worse results; a few representative ones (only, due space limitations) are

| Solver | e. err | b/ml ratio | $\mathrm{ml} / \mathrm{d}$ ratio | opt. features |
| :---: | :---: | :---: | :---: | :--- |
|  | 0.548 | 0.775 | 0.969 | $[0,1]$ |
| RelaxIV | 0.479 | 0.755 | 0.975 | $[0,1,18,21,23]$ |
|  | 1.224 | 0.768 | 0.996 | $[0,1,3,7,11,15,20,23]$ |
|  | 0.513 | 0.862 | 0.923 | $[0,1]$ |
| MCFSimplex | 0.720 | 0.867 | 0.928 | $[0,1,4,8,12,16,24]$ |
|  | 1.472 | 0.852 | 0.953 | $[0,1,3,7,11,15,20,23]$ |

Using the selected features for each solver, we applied the ML to determine the solver to be used. In the following table we report the percentage of solver mis-selection ("s. err"), the ratio between
the running time of the actual best solver (with its best configuration) and that of the (Solver, Configuration) chosen by the ML ("b/ml ratio"), and the ratio between the latter and the default configuration for both solvers ("ml/dR ratio" and " $\mathrm{ml} / \mathrm{dS}$ ratio", respectively).

| s. err | $\mathrm{b} / \mathrm{ml}$ ratio | $\mathrm{ml} / \mathrm{dR}$ ratio | $\mathrm{ml} / \mathrm{dS}$ ratio |
| :---: | :---: | :---: | :---: |
| 0.145 | $0.787(0.208)$ | $0.933(0.342)$ | $0.791(0.450)$ |

### 6.3.1 Further experimentation

Following the same methodology used in the previous section, we conducted further experiments to measure the potentiality of a more complex set of instances and a bigger set of solvers for the case of the MCF problem. Moreover we conducted other experiments with NLP problems and MIP problems.

For the MCF case, we used a slightly smaller set of instances, but fairly more complex than the previous one. We created 132 instances using the generators: complete networks (500-3000 nodes $/ 310991-20247330$ arcs), gridgen (256-65536/8192-1048576), netgen (1024-16384/32768-1048600) and goto (512-65536 nodes/32768-4194304 arcs).

The solvers used were,

1. cplexbarrier - CPLEX Barrier Optimizer to solve large, sparse linear programming problems [7],
2. cplexnet - CPLEX Network Optimizer on linear programming problems based on a network model [7],
3. MCFSimplex - a recent implementation of the classical network simplex algorithm [19],
4. LEMON_capacityscaling - Capacity Scaling: dual method, which can be viewed as a generalization of the Ford-Fulkerson algorithm [62],
5. LEMON_netsimplex - Network Simplex: a specialized version of the linear programming simplex method [57],
6. LEMON_costscaling - Cost Scaling: a primal-dual approach, which can be viewed as the generalization of the push-relabel algorithm [79],
7. LEMON_cyclecanceling - Cycle Canceling: a general primal method [103],

Solvers 1-3 were distributed by the MCFClass project [124], and 4-5 were distributed by the COIN-OR::LEMON project [14].

Since the instances were more complex, some of the solvers did not provide a solution for Dconf (and sometimes for any configuration). Therefore we will report three new statistics: "ns" the amount of instances not solved with any configuration, "ns-D" the amount of instances not solved with Dconf.

| Solver | ns | ns-D | nbest | b/d ratio | $\%$ best |
| :---: | :---: | :---: | :---: | :---: | :---: |
| cplexbarrier | 30 | 0 | 13 | $0.886(0.188)$ | 0.0 |
| cplexnet | 4 | 3 | 25 | $0.908(0.134)$ | 2.2 |
| MCFSimplex | 4 | 0 | 28 | $0.841(0.207)$ | 26.5 |
| LEMON_capacityscaling | 8 | 0 | 7 | $0.849(0.236)$ | 3.78 |
| LEMON_costscaling | 4 | 0 | 7 | $0.922(0.151)$ | 2.2 |
| LEMON_cyclecanceling | 8 | 0 | 1 | $1.0(0.0)$ | 0.0 |
| LEMON_netsimplex | 4 | 3 | 8 | $0.583(0.329)$ | 65.1 |

We also mention that the best solver requires on average around $3.77 \%$ of the running time of the worst solver.

Considering a wider set of solution methods (plus more complex instances) we emphasize even more the need of taking into account different configurations and different solvers. Assume that we just use the solver LEMON_netsimplex (the solver with best performance) we will be solving
the $34.9 \%$ of the instance with a solver that is not the best (and might be the worst). Moreover, observe how LEMON_netsimplex, even if it is the most used solver, owes its performance to the fact the we considered the Aconf. If we considered the Dconf, its performance would have decayed by more than a $40 \%$.

We also did some experiments for MIP and NLP problems. For the MIP problems we considered 93 instances from MIPLIB 2003 [17] and the CPLEX solver distributed by GAMS [15]. For the NLP problems we considered 415 instances from Global Library [6] and the snopt and conopt solvers [5] also distributed by GAMS.

Since the solvers for MIP and NLP may not always converge to a global optima, we will consider new statistics related with the best objective function (OF) value, like: "(b/d) of ratio" the average ratio between the Aconf OF value - Dconf OF value and |Dconf OF value|; "bof/dof ratio" the average ratio between AOFconf OF value - Dconf OF value and |Dconf OF value| (where AOFconf if the configuration that provided the best OF value); and "(bof/dof) ${ }^{t}$ ratio" the average ratio between AOFconf time and Dconf time.

| Type | Solver | ns | ns-D | nbest | b/d ratio | $(\mathrm{b} / \mathrm{d})^{\text {of }}$ ratio | bof/dof ratio | (bof/dof) ${ }^{\mathrm{t}}$ ratio | \% best | \% best OF |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| MIP | CPLEX | 22 | 2 | 27 | $0.730(0.311)$ | $0.0(0.0)$ | $0.0(0.0)$ | $0.730(0.311)$ | - | - |
|  | snopt | 58 | 12 | 54 | $0.520(0.322)$ | $2.897(28.092)$ | $-0.025(0.173)$ | $0.740(0.939)$ | 27.6 | 9.2 |
| NLP | conopt | 18 | 27 | 72 | $0.712(0.314)$ | $0.274(2.921)$ | $-0.017(0.125)$ | $1.148(3.039)$ | 72.3 | 90.7 |

Note, mainly for the NLP case, that the OF values difference ratios are sometimes a negative number, this means that there was a gain (ie. the AOFconf OF value was better than the Dconf OF value). Also for NLP the best solver requires on average around $48.5 \%$ of the running time of the worst solver, and the best OF value improves in $18.3 \%$ the worst one.

### 6.4 Discussion

In this chapter we described the set of architectural choices in the I-DARE system that have been designed to make an effective search, while avoiding to tie-in the system to specific search strategies that may not ultimately prove effective enough. In particular, we discussed the fundamental role of the GMLS, which allows to integrate general-purpose ML approaches with specialized methods for the nontrivial task of computing $\psi$. This task is "naturally" extended to that of selecting the best algorithmic configuration of the available solvers, thereby providing the I-DARE(control) sub-system with a powerful tool to streamline the search. This requires a sophisticated ML (meta) process that is continuously running and keeps modifying the assessment of each reformulation with respect to given algorithms. Although use of ML techniques to select algorithmic parameters have very recently been advocated elsewhere, the scale of our proposal is, to the best of our knowledge, unheard of.

I-DARE(control) defines a central component d_control that manages all the search process. This component can potentially implements any known search technique, from Heuristics and Meta-heuristics (GRASP, Simulated Annealing, Taboo Search, Genetic Algorithms, Ant Colony, etc) to enumeration algorithms (like Branch and Bound). It is quite clear for us that the selection of a proper search technique (or combination of search techniques) is a big issue. This is why we designed the system avoiding the limitation of choosing a specific search technique.

A feature of the system is that the reformulations made to the model immediately impact which solver(s) will be used, and therefore the specific options that may be chosen in a solver. For instance, if a model is reformulated to a MILP with a specific structure, for which specific cuts can be generated, then a solver attached to that structure may (and likely will) have the option to generate (or not) these cuts. Moreover, for MINLP problems some algorithms may use different combinations of solvers to solve the LP and NLP problems [56]. The I-DARE environment not only allows the choice of different combinations, but also automatizes the selection of the "best" choice for any given instance, which has the possibility to significantly increase the performances of the approach. I-DARE prefixes the possibility of dealing with a vast set of mathematical models, from
large-scaled structured LP/MILP, to NLP/MINLP and more exotic classes like PDE constraints [89].

The results attained with the experiments prove that even if the training and testing sets were rather small, the usage of the ML technique, applied to select the solver and configuration, improved indeed the average time with respect to Dconf. Meaning that the default configuration we may find in solvers we use, it is not always the best choice, due to the heterogeneity of the instances. Moreover the features to be used by each solver in the ML process may no be the same (for solvers attached to the same structure). The training and Meta-Learning processing time depends on: the size and complexity of the problems being solved for generating the training points; the amount of instance features and configuration fields of each point; and even the amount of parameters the MLs may have.

The outcome of this sophisticated process may well be a very significant improvement of the efficiency experienced by the "average" (non expert) user in the solution of her models, thereby significantly contributing to the overall scientific and technological progress.

## Chapter 7

## Combining structures and reformulations


#### Abstract

In this chapter we will declare a set of "simple" structure classes that will allow (by composition) the creation of complex models. Even though, most of these structure class may not have a solver attached, they will become useful in the reformulation process. We will define some $\mathrm{ARR}^{\Sigma_{\mathrm{S}}}$ in order to transform these structures classes into d_MILP_C (or d_LP_C), and finally, we will focus on a particular non linear model (that uses these structure classes) and we will see how by applying the defined $\operatorname{ARR}^{\Sigma_{S}}$ we can transform this model from a MINLP formulation into an equivalent MILP reformulation.


### 7.1 Structures

One of the main I-DARE potentialities is the capacity of declaring and relating structures that contain a specific semantic value. In this section we will focus on creating a set of global structures that will allow us to build models by combining them.

For instance we may declare some simple structures just to define a binary variable (BV), continuous variable (CV), relation and a constant.

```
d_SingleBV_C :: d_LeafProblem_C
[
    args -> [ v = d_var ]
].
d_SingleCV_C :: d_LeafProblem_C
[
    args -> [ v = d_var ]
].
d_Relation_C :: d_LeafProblem_C
[
    args -> [rel = d_rel]
].
d_Constant_C :: d_LeafProblem_C
[
    args -> [c = d_constant]
].
```

We may also define, for example a vector of continuous variables,

```
d_VectorCV_C :: d_LeafProblem_C
[
    dim_var -> [D],
    args -> [ v = d_vector(d_var, [D]) ]
].
```

Considering more complex structures, we can create for instance a product between a CV and a BV,

```
d_ProdBC_C :: d_Block_C
[
    ids -> [bin , cont ],
    subsC -> [d_SingleBV_C, d_SingleCV_C],
    link -> [([X],d_all) , ([Y], d_all)]
    rpIR -> [bin = 1, cont = 1]
].
```

Moreover we can declare a structure to represent a semi-continuous expression, like $f * x$, where $f$ is a continuous structure (i.e. using only CVs ) and $x$ is a BV .

```
d_SemiContinuous_C :: d_Block_C
[
    ids -> [ct [d_Component_C, bv_SingleBV_C],
    link }->\mathrm{ [([X], d_all), ([Y], d_all)],
    rplR }->\mathrm{ [ct = 1, bv = 1]
].
```

Observe we do not explicitly verify that ct is a continuous expression. However, since any reformulation declared from or to d_SemiContinuous_C will use a narrowing, then at that point we will specify only continuous structures. We will see that since we will not assign a solver to most of the structures within this chapter, their only meaningful functionally will be the definition of reformulation. Considering other operators like, $|\cdot|$ (absolute value), we can create further structures.

For instance the following one, represents $\left|\sum_{i} v_{i} c_{i}\right|$, where $v_{i}$ is a CV and $c_{i}$ is a constant,

```
d_VAbs_C :: d_LeafProblem_C
[
    dim_var -> [D],
    args -> [
        v = d_vector(d_var, [D]),
        c = d_vector(d_constant, [D])
            ]
].
```

Or we could make the non-vectorial version of the previous class,

```
d_SAbs_C :: d_LeafProblem_C
[
    args -> [
        v = d_var,
        c = d_constant
            ]
].
```

Structures representing specific collections of constraints and/or optimization problems can also be defined, like d_LP_C (cf. Listing 1.5); d_MILP_C (cf. Listing 2.23); the semi-assignment constraints (cf. §4.1.2),

```
d_SemiAssign_C :: d_LeafProblem_C
[
    dim_var -> [D],
    args -> [
```

```
    v = d_vector(d_var, [D])
    ]
].
```

and the complementary constraints defined by $x y=0$ where $x, y \geq 0$ are CVs.

```
d_ProdCC_C :: d_LeafProblem_C
[
    args -> [
        x = d_var,
        y = d_var
    ]
].
```

Beside those specific structures we can define a structure to represent a general constraint $\mathrm{f}=</=/\rangle=\mathrm{c}$, where c is a constant, and f can be any component,

```
d_Constraint_C :: d_Block_C
[
    ids -> [expr , rel c, c],
    subsC -> [d_Component_C, d_Relation_C, d_Constant_C],
    link ->> [([X], d_all), ([],d_all) , ([], d_all)],
    rpIR -> [expr=1, rel = 1,c=1]
].
```

Note that d_Relation_C and d_Constant_C are helper structures to put a single relation or a constant inside a block. Also, observe that if expr (as well as rel and c) has free indices, they must be equal to the free indices in the constraint. Therefore no internal replication is allowed.

Likewise, we can create a structure to represent a general minimization objective function,

```
d_OFMin_C :: d_Block_C
[
    ids -> [expr ],
    subsC -> [d_Component_C],
    link -> [([X], d_all) ],
    rpIR -> [expr = 1
```

].

Like for d_Constraint_C, no internal replication is allowed in d_OFMin_C.

### 7.1.1 Compositions

Once we have the single structures we may want to compose them to obtain more complex structures. The following structure combines to general structures that share a set of variables,

```
d_Composition_C :: d_Block_C
[
    ids }->\mathrm{ [p1 , p2 [ [ ],
    subsC -> [d_Component_C , d_Component_C ]
    link -> [([X,Y], d_all), ([X,Z], d_all)],
    rpIR -> [p1 = 1, p2 = 1]
].
```

Observe that both substructures share a set of variables $(X)$ and have independent sub-set of variables ( Y and Z ).

Another composition case can be based on the internal replication of a sub-structure.

```
d_IndComposition_C :: d_Block_C
[
    ids }->\mathrm{ [s ],
    subsC -> [d_Component_C],
    link -> [([X], d_all)]
].
```

Notice that the internal structure s can be replicated inside of d_IndComposition_C, implying that each replication will have an independent set of variables. Therefore, the substructures are completely separable. This fact will prove useful during reformulations, while integrating particular narrowings of d_IndComposition_C. We can specify a general behavior by saying that d_IndComposition_C will sum all isolated terms and concatenate all constraints.

### 7.2 Creating a model

In this section we propose the representation of a Hyperplane Clustering Problem (HCP). In a HCP we have a set of points $p=\left\{p_{i} \mid i \in M\right\} \in \mathbb{R}^{D}$ and we want to find the set of $N$ hyperplanes $w=\left\{w_{j 1} x_{1}+\ldots+w_{j d} x_{d}=w_{j}^{0} \mid j \in N\right\} \in \mathbb{R}^{D}$ and an assignment of points to hyperplanes such that the distances from the hyperplanes to their assigned points are minimized. HCP can be algebraically defined by the following MINLP,

$$
\begin{align*}
\min & \sum_{i \in M} \sum_{j \in N}\left|w_{j} p_{i}-w_{j}^{0}\right| x_{i j}  \tag{7.2.1}\\
\text { s.t. } & \sum_{j \in N} x_{i j}=1 \quad \forall i \in M  \tag{7.2.2}\\
& \sum_{k \in D}\left|w_{j k}\right|=1 \quad \forall j \in N  \tag{7.2.3}\\
& w \in \mathbb{R}^{N \times D}, \quad w^{0} \in \mathbb{R}^{N}, \quad x \in\{0,1\}^{M \times N}
\end{align*}
$$

Note HCP has a parameter $p \in \mathbb{R}^{M \times D}$, and dimensions $N, M, D \subset \mathbb{N}$. Differently to previously seen examples, HCP introduces nonlinearity in both the objective function and constraints.

To model HCP we will use a combination of the previously specified structures. Note that (7.2.1) is an objective function containing products between absolute values and BVs ; (7.2.2) is a semi-assignment; and (7.2.3) is a constraint containing absolute value operations. Hence, we can build the following model.

## Dimensions, indices and Properties

```
d_dimension(D). d_dimension(N). d_dimension(M).
d_index(i, M). d_index(j, N). d_index(k, D).
    d_constant
p : d_property
dims }->\mathrm{ [M, D]
].
    d_var.
w : d_property
    dims -> [N, D]
].
```

```
w0 : d_var.
w0 : d_property
[
    dims -> [D]
] .
x : d_var.
x : d_property
[
    dims -> [M, N],
    lower -> 0,
    upper -> 1
].
```


## Structures

```
vabsof : d_VAbs_C
[
    args -> [ // freeinds = (i,j)
                        v=$([$(w(j,k),[k]),w0(j)]),
            c=$([$(p(i,k),[k]), 1 ])
        ]
].
bvof : d_SingleBV_C
[
    args -> [ // freeinds = (i,j)
        v = x(i,j)
    ]
].
semicof : d_SemiContinuous_C
[
    subs -> [absof , bvof],
    subVP }->\mathrm{ [[(w,w0)], [x]],
    freel -> [i,j]
].
indof : d_IndComposition_C
[
    subs -> [semicof ],
    subVP -> [[(w,w0,x)]]
].
of : d_OFMin_C
[
    subs -> [indof ] ],
].
semiac : d_SemiAssign_C
[
    args -> [ // freeinds = i
        v = $(x(i,j),[j])
].
sabsc : d_SAbs_C
[
    args -> [ // freeinds = (j,k)
        v = w(j,k),
        c=1,
    ]
].
rel : d_Relation_C
[
    args -> [rel = '=']
].
```

The diagram in Figure 7.1 shows the HCP formulation by representing only the name and class of the structures used, plus the relations between them

### 7.3 Reformulations

In this section we will introduce some of the reformulations that can be created based on the previously defined structures. Our main goal in this case will be trying to remove the non linearity elements in a reformulation, by adding the proper additional variables and constraints. We will use the classes d_MILP_C and d_LP_C (cf. Listings 2.23 and 1.5) as the main goals in the reformulation rules to be presented herein. Most of the reformulation rules exposed in this section were extracted


Figure 7.1: HCP Formulation
from [112].
In some the cases, the generated MILP and LP will have no objective function (i.e. all the cost constant will be 0 ), so we will not specify the direction parameter, because it is irrelevant. In other cases, when integrating two MILPs, for instance, we will use the fact that the d_direction type is evaluated as 1 if equal to min and -1 if equal to max. So depending on the unified direction we want to arrive, we will transform the cost constants of the objective function.

### 7.3.1 ProdBC to MILP

A product between a BV $b$ and a CV $x \in[0 . . U]$, can be substituted by a continuous variable $w \in[0 . . U]$ and the constraints: $w-U b \leq 0, w-x \leq 0$ and $x+U b-w \leq U$. So we can build the following $\mathrm{ARR}^{\Sigma}$.

```
d_ProdBC_to_MILP_ARR : d_ARR_AIgebraic
[
    A \(\rightarrow\) d_ProdBC_C(?_, ? ) ,
    B \(\rightarrow\) d_MILP_C,
    indexA \(\rightarrow[1=(i, i 1), 2=j]\),
    indexB \(\rightarrow\) [],
    \(\operatorname{dimRel} \rightarrow[\) colsl \(=1\), colsR \(=2\), cons \(=3\) ],
    arg_map \(->\) [
        B..ci \(=0\),
        \(B \ldots c R=\$([0,1])\),
        \(B . . A i=\$([\)
            \([\$(-1 * \operatorname{up}(A . . c o n t \ldots v),[i 1, i])]\),
                                    \$( 0, [i1, i]) ],
                            [ \$( _up(A..cont..v) , [i1, i]) ]
            B. . Ar \(=\$([\)
                    \([\$(\operatorname{ccs}([1->j=1,0]),[i 1, j])]\),
                    \([\$(\cos ([1->j=1,-1]),[i 1, j])]\),
```

```
                        [ $(-cs([1->j=0, -1]), [i1,j]) ]
                ]),
    B..rels = '=<'
    B..b = $([0, 0, -up(A..cont..v)]),
    B...xi = lower(0),
    B..xi = upper(1),
    B...xr = lower (0),
    B..xr = upper(-up(A..cont..v)),
    B...xr = [v=1, aux=1]
    ],
    A..bin..v = B..xi
    A..cont..v = B..xr(v)
    ]
].
```

Note that the objective function of the generated MILP has a non-zero constant for the variable that must substitute $b x$, and the rest of the constants are 0 . The utility of this objective function constants, will be seen later, when reformulating d_OFMin_C and d_Constraint_C.

### 7.3.2 SAbs to Composition

If we consider a structure involving a term $|p v|$ (d_SAbs_C, $p$ is a constant and $v$ is a CV), this term can be reformulated so that it is differentiable, by adding two CVs $t^{+}, t^{-} \in[0 . .+\infty]$; replacing $|p v|$ by $t^{+}+t^{-}$; and adding the constraints $p v-t^{+}-t^{-}=0$ and $t^{+} t^{-}=0$. This reformulation involves a linear substructure, plus a complementary constraint $(x y=0)$. So we can define an $\operatorname{ARR}^{\sum}$ that transforms d_SAbs into a composition between a d_LP_C and a d_ProdCC_C.

```
d_SAbs_to_Composition_LP_ProdCC_ARR : d_ARR_Algebraic
[
    A \(\rightarrow\) d_SAbs_C,
    B \(\rightarrow\) d_Composition_C(d_LP_C, d_ProdCC_C),
    indexA \(\rightarrow\) [1=i, \(3=j]\),
    indexB \(\rightarrow\) [],
    dimRel \(\rightarrow\) [p1..cols=3, p1..cons=1],
    arg_map \(\rightarrow\) [
        B..p1..c \(=\$([0,1,1])\),
        \(B \ldots p 1 \ldots A=\$(-\operatorname{cs}([A \ldots p->(j=0),-1]), \quad[i, j])\),
        B..p1...rels \(=\) ' \(=\) ',
        B..p1..b \(=0\),
        B..p1.. \(x=[w=1, t p=1, t m=1]\),
        B..p1...x(tp) \(=\) lower(0),
        B..p1...x(tm) \(=\) lower (0)
        B..p1..xi \(=\) lower (0),
        B..p1...xi \(=\) upper (1)
            ],
    ans_map -> [
            A..w \(=B . . \times(w)\),
    fixTI \(\rightarrow \stackrel{]}{[ }\)
    \(X=(B \ldots p 1 \ldots x(t p), B \ldots p 1 \ldots x(t m))\),
    \(Y=B \ldots p 1 \ldots \times(w)\),
    Z = []
    ]
].
```

Notice that the substitution of $|p v|$ is expressed by defining the c constants in d_LP_C with 0 for $v$ and 1 for $t^{+}$and $t^{-}$.

### 7.3.3 VAbs to LP

Considering now a term $\left|\sum_{i} p_{i} v_{i}\right|$ we can apply a similar reformulation to the one defined in the previous section. However in this case we will consider that the term is inside a minimization
function (the same way can be done for d_SAbs_C). In this case, the complementary constraint can be eliminated because we are minimizing $t^{+}+t^{-}$, so due to the function's direction, at a global optimum, one of $t^{+}$or $t^{-}$will have value zero. Therefore implying the complementary constraint.

Hence, in this case, we will reformulate d_VAbs_C into an d_LP_C,

```
d_VAbs_to_LP_oncond_OFMin_ARR : d_ARR_Algebraic
[
    A -> d_SAbs_C,
    B }->\mathrm{ d_LP_C,
    indexA -> [2=i, 1=j, D=d],
    indexB }->\mathrm{ [],
    dimRel }->\mathrm{ [cols=D+2, cons=1],
    arg_map -> [
        B..c = $([ $(0, [d]), $(1, [i])]),
        B..A = $([
                        [$(A..c(d), [j,d]),
                        $(-1 , [j,i])
                    ]),
            B..rels = '=',
            B..b = 0,
            B..x = [v=[D], t=[2]],
            B...x(v) = lower(_Iw(A..v)),
            B..x(v) = upper(-up(A..v)),
            B..x(t) = lower(0),
            ];
            A..v = B..x(v)
            ],
    condition -> _parent(A) = d_OFMin_C
].
```

Notice the condition inside the $\mathrm{ARR}^{\Sigma}$ indicating that A must have a parent d_OFMin_C inside the block's tree, thus implying it is inside a minimization function.

### 7.3.4 SemiContinuous to MILP

If we manage to narrow a d_SemiContinuous_C until the point of knowing that it has an d_LP_C inside, then we can easily transform d_SemiContinuous_C into a d_MILP_C. Assume the LP has the form

$$
\begin{aligned}
& \min c^{T} x \\
& \text { s.t. } A x=b \\
& \quad x_{i} \in\left[0 . . B_{i}\right]
\end{aligned}
$$

then the fact of multiplying this LP by a BV $y$ (only in the objective function) creates the following MINLP

$$
\begin{aligned}
\min & \sum_{i} c_{i} x_{i} y \\
\text { s.t. } & A x=b \\
& x_{i} \in\left[0 . . B_{i}\right], \quad y \in\{0,1\}
\end{aligned}
$$

This MINLP can be reformulated into a MILP by applying the same mechanism used for d_ProdBC_to_MILP_ARR (cf. §7.3.1). We may add a CV $w_{i} \in\left[0 . . B_{i}\right]$ to substitute each product $x_{i} y$, and then add the constraints $w_{i}-B_{i} y \leq 0, w_{i}-x_{i} \leq 0$ and $x_{i}+B_{i} y-w_{i} \leq B_{i}$. Resulting in the following MILP,

$$
\begin{aligned}
& \min \sum_{i} w_{i} \\
& \text { s.t. } A x=b
\end{aligned}
$$

$$
\begin{aligned}
& w_{i}-B_{i} y \leq 0 \quad \forall(i) \\
& w_{i}-x_{i} \leq 0 \quad \forall(i) \\
& x_{i}+B_{i} y-w_{i} \leq B_{i} \quad \forall(i) \\
& w_{i}, x_{i} \in\left[0 . . B_{i}\right], \quad y \in\{0,1\}
\end{aligned}
$$

Hence, we can create the following $\operatorname{ARR}^{\Sigma}$ to represent this reformulation,

```
d_SemiContinuous_LP_SingleBV_to_MILP_ARR : d_ARR_AIgebraic
[
    A }->\mathrm{ d_SemiContinuous_C(d_LP_C, ?_),
    B -> d_MILP_C
    indexA -> [cols=(dr,dr1), 1=ii, cons=vj],
    indexB -> [],
    dimRel -> [colsl=1, colsR=2*cols, cons=cons+3*cols],
    arg_map -> [
            B..cl = 0,
            B..cR = $([$(0, [dr]), $(1, [dr])]),
            B..Ai = $([ // int matrix (cons+3*cols x 1)
                                    [ $(-1*_up(A..ct ..x(dr)), [dr, ii])],
                                    [$(0 , [dr,ii])],
                            [$(_up(A..ct..x(dr)) , [dr,ii]) ],
                            [ $(0
            B..Ar = $([]// real matrix (cons+3*cols x 2*cols)
                    $(_cs([A..ct .cc(dr1)->(dr1=dr),0]), [dr,dr1]),
                            $(0 , [dr,dr1])
                            ],
                            $(_cs([A..ct_c(dr1)->(dr1=dr),0]), [dr,dr1]),
                    $(-cs([-1->(dr=dr1),0])},[dr,dr1]
                            ] [
                            $(_cs([-1*A..ct ..c(dr1)->(dr1=dr),0]), [dr,dr1]),
                    $(_cs([1-> (dr=dr1),0]) , [dr,dr1])
                            ],
                            $(A..ct..A(vj,dr1), [vj,dr1]),
                    $(0 , [vj,dr1])
            B..rels= [ $),
                $('=<', [dr]),
                $('=<', [dr]),
                $(A..ct..b(vj), [vj])
            B..b = $([)
                $(0, [dr]),
                $('=<', [dr])
                        $(A..ct..rels(vj), [vj])
                            ]),
            B..xr = [o=[cols], n=[cols]],
            B..xi = lower(0),
            B..xi=upper(1),
            B..xr(o)= lower(_lw(A..ct..x)),
            B..xr(o)= upper(-up(A..ct..x)),
            B...xr(n)= lower(0),
            B..xr(n)= upper(_up(A..ct..x)),
            ],
            A..bv..v = B..xi,
            A..ct\ldotsx = B..xr(o)
    ]
].
```


### 7.3.5 ProdCC to MILP

When in presence of a complementary constraint $x y=0$, we can substitute it by the following MILP constraints, $x-M z \leq 0$ and $y+M z \leq M$, where $z \in\{0,1\}$ and $M$ is a sufficiently large number.

```
d_ProdCC_to_MILP_ARR : d_ARR_Algebraic
[
    A \(\rightarrow\) d_ProdCC_C,
    \(B \rightarrow\) d_MILP_C,
    indexA \(\rightarrow\) [1=(i,i1), 2=j],
    indexB \(\rightarrow\) [],
    \(\operatorname{dimRel} \rightarrow[\) colsl \(=1\), colsR \(=2\), cons \(=2\) ],
    arg_map \(\rightarrow\) [
            B..ci \(=0\),
            B.. \(c R=0\),
            B..Ai \(=\$([\)
                            [ \$(-1*_big, [i1, i]) ],
                    \([\$(\) _big , [i1, i] \()]\)
            B.. Ar \(=\$([\) )
                                    \(\left.\begin{array}{ll}\left.\left[\begin{array}{ll}\$(-\operatorname{cs}([1->j=0, & 0]),\end{array}[i 1, j]\right)\right], \\ {[(-\csc ([1->j=1,} & 0]),\end{array}\right]\)
                    ]),
            B..rels \(=\) ' \(=<^{\prime}\),
            B..b \(=\$([0, \quad-b i g])\),
            B...xi \(=\) lower (0),
            \(B \ldots x i=u p p e r(1)\),
            B.. \(x r=[x=1, y=1]\),
            B...xr \((x)=\) lower \((-\operatorname{lw}(A \ldots x))\),
            B..xr \((x)=\operatorname{upper}(-u p(A \ldots x))\)
            B...xr \((y)=\) lower ( \(\operatorname{lw}(A \ldots y))\),
            \(B \ldots \operatorname{xr}(y)=\operatorname{upper}(\operatorname{up}(A \ldots y))\)
            ],
    ans_map \(\rightarrow\);
            A.. \(x=\) B.. \(x r(x)\),
            \(A \ldots y=B \ldots x r(y)\)
            ]
].
```

Observe that since d_ProdCC_C represents a constraint, the generated MILP has no objective function.

### 7.3.6 SemiAssign to MILP

The semi-assignment constraint $\sum_{i} y_{i}=1$, has trivial transformation into a MILP with no CVs,

```
d_SemiAssign_to_MILP_ARR : d_ARR_AIgebraic
[
    A -> d_SemiAssign_C,
    B }->\mathrm{ d_MILP_C,
    indexA -> [1=j, D=d],
    indexB -> [],
    dimRel -> [colsl=D, colsR=0,
                    cons=1],
    arg_map -> [
            B..ci = 0
            B..cr = 0
            B..Ai = $(1, [j, d]),
            B..rels = '=',
            B..b = 1,
            B..xi = lower(0),
            B..xi = upper(1)
            ],
            A..v = B...xi
```

```
].
```


### 7.3.7 Constraint to MILP

Having a d_Constraint_C with its substructure narrowed to a MILP, allows us to transform the whole constraint structure into a MILP. We will assume that the objective function $\left(\sum_{i} c_{i} x_{i}\right)$ of the inner MILP will represent, regardless of its direction, a last row of the LHS matrix of the new generated MILP. This last row is obtained by combining $\sum_{i} c_{i} x_{i}$ with the d_Relation_C and d_Constant_C substructures of d_Constraint_C. Therefore the resulting MILP will include all constraints of the inner MILP plus $\sum_{i} c_{i} x_{i}$ d_rel d_constant.

The following $\mathrm{ARR}^{\Sigma}$ reformulates d_Constraint_C(d_MILP_C ? _- ?-) into a d_MILP_C,

```
d_Constraint_MILP_to_MILP_ARR : d_ARR_AIgebraic
[
    A \(\rightarrow\) d_Constraint_C(d_MILP_C, ?_, ? \()\),
    \(B \rightarrow\) d_MILP_C,
    indexA \(\rightarrow\) [1=j1, expr..colsl=ii, expr..colsR=ir, expr..cons=j],
    indexB \(\rightarrow\) [],
    dimRel \(\rightarrow\) [colsl=expr..colsl, colsR=expr..colsR,
                    cons=expr..cons +1 ],
    arg_map \(->\) [
        B.. ci \(=0\)
        B..cr \(=0\)
        \(\mathrm{B} . \mathrm{Ai}=\$([/ /\) LHS int matrix (cons \(+1 \times\) colsl)
                            [ \$(A..expr..Ai(j, ii), [j, ii])],
                            \$(A..expr..ci(ii) , [j1,ii])]
                ]),
            B..Ar \(=\) ([ // LHS real matrix (cons* \((F I+1) \times \operatorname{colsR} * F I)\)
                    [ \$(A..expr..Ar \((\mathrm{j}, \mathrm{ir}),[\mathrm{j}, \mathrm{ir}])\) ],
                [ \$(A..expr..cr(ir) , [j1,ir])]
                ]) ,
                        \$(A..expr.. rels(j), [j]),
                            \$(A..rel..rel , [j1])
                B..b \(=\$([\)
                                    \$(A..expr..b(j), [j]),
                                    \$(A..c.cc \(\quad[j 1])\)
                            ]),
                B..xi \(=\) lower (_lw (A..expr...xi))
                B..xi \(=\operatorname{upper}(\operatorname{up}(A \ldots \operatorname{expr} \ldots x i))\)
                B...xr \(=\operatorname{lower}(\ldots \mathrm{lw}(A \ldots\). expr...xr)) ,
                \(B \ldots x r=\operatorname{upper}(\operatorname{up}(A \ldots \operatorname{expr} \ldots x r))\)
            ] ,
            A..expr..xi \(=\) B.. \(x i\),
            A..expr..xr \(=\) B...xr
            ]
].
```

Notice that an $A R R^{\Sigma}$ to reformulate d_Constraint_C(d_LP_C, ?_, ?_) into d_LP_C can be created in an analogous way.

### 7.3.8 OFMin to MILP

The reformulation of a d_OFMin_C with the inner structure narrowed to a MILP is even more direct that the d_Constraint_C case, because the objective function is left as it is, except for the sign transformation depending on the inner MILP direction.

```
d_OFMin_MILP_to_MILP_ARR : d_ARR_AIgebraic
[
    A -> d_OFMin_C(d_MILP_C),
```

```
    B -> d_MILP_C,
    indexA \(\rightarrow\) [expr..colsl=ii, expr..colsR=ir, expr..cons=j],
    indexB \(\rightarrow\) []
    dimRel \(\rightarrow\) [colsl=expr..colsl, colsR=expr..colsR,
    cons=expr..cons],
    arg_map -> [
        B..ci \(=\) (A..expr..dir*A..expr..ci(ii), [ii]),
        B..cr \(=\$(A \ldots\) expr..dir*A..expr...cr(ir), [ir]),
        \(B . . A i=\$(A \ldots \operatorname{expr} \ldots A i(j, i i),[j, i i])\),
        \(B . . A r=\$(A \ldots\) expr..Ar(j,ir), \([j, i r])\),
        B..rels \(=\$(\) A..expr..rels \((j),[j])\),
        B..b \(=\$(A \ldots\) expr..b(j), [j]),
        B..dir \(=\) 'min'
        B...xi \(=\operatorname{lower}(\operatorname{llw}(A .\). expr..xi)),
        B...xi \(=\operatorname{upper}(-\operatorname{up}(A \ldots\) expr..xi)),
        B..xr \(=\operatorname{lower}(-\operatorname{lw}(A \ldots\) expr..xr)) ,
        B...xr \(=\operatorname{upper}((\operatorname{up}(A . . \operatorname{expr} \ldots \times r))\)
        ],
        A..expr...xi \(=\) B...xi,
            A..expr...xr \(=B . . \times r\)
        ]
].
```

Again in this case the reformulation from d_OFMin(d_LP_C) to d_LP_C can be done in an analogous way.

### 7.3.9 IndComposition to MILP

The d_IndComposition_C structure with the inner structure narrowed to MILP, can be reformulated into a single MILP, by mixing the inner replicated structures. For instance if the inner MILP has a free index $j$ then each MILP ${ }^{j}$ has an independent set of variables with respect to the other MILP ${ }^{j^{\prime}}$, with $j \neq j^{\prime}$. Therefore the resulting MILP can be composed as shown in Figure 7.2.


Figure 7.2: Independent Composition of $N$ MILP subproblems
The $c^{j}$ constants will be multiplied by the direction of $\mathrm{MILP}^{j}$ in order to unify the objective function to a minimization.

Applying this composition we can define the following $\mathrm{ARR}^{\Sigma}$ to reformulate a d_IndComposition_C(d_MILP_C) into a single d_MILP_C.

```
d_IndComposition_MILP_to_MILP_ARR : d_ARR_AIgebraic
[
    A }->\mathrm{ d d_IndComposition_C(d_MILP_C)
    B -> d_MILP_C,
    indexA -> [_fiD(s)=(fi,fi1), 1=i,
    s..colsl=ii, s..colsR=ir, s..cons=j],
    indexB -> [],
    dimRel -> [colsl=s..colsl*_fiD(s), colsR=s..colsR*_fiD(s),
```

```
    cons=s..cons*_fiD(s)],
    arg_map -> [
    B..ci = $(A..expr(fi)..dir*A..expr(fi)..ci(ii), [(ii,fi)]),
    B..cr = $(A..expr(fi)..dir*A..expr(fi)..cr(ir), [(ir,fi)]),
    B..Ai = $(_cs([A..expr(fi)..Ai(j,ii)->(fi=fi1), 0]),
                [(j,fi), (ii,fi1)]),
            B...Ar = $(-cs([A...expr(fi)..Ar(j,ir)->(fi=fi1), 0]),
                [(j,fi), (ir,fi1)]),
            B..rels = $(A..expr(fi)..rels(j), [(j,fi)]),
            B..b = $(A..expr(fi)..b(j), [(j,fi)]),
            B..dir = 'min',
            B..xi = [f=_rpl(_fiD(expr), [expr..colsl])],
            B..xr = [f=-rpl(_fiD(expr), [expr..colsR])],
            B..xi(f(fi)) = lower(_lw(A..s(fi)..xi)),
            B_..xi(f(fi))=upper(-up(A..s(fi)..xi)),
            B...xr(f(fi)) = lower(-lw(A..s(fi)..xr)),
            B..xr(f(fi)) = upper(-up(A..s(fi)..xr))
            > ],
            A..s(fi)...xi = B...xi(f(fi)),
            A..s(fi)..xr = B..xr(f(fi))
    ]
].
```

We could define a similar reformulation to integrate several d_LP_C into a single d_LP_C

### 7.3.10 Composition to MILP

When the composition of two structures, with shared variables (d_Composition_C), has both substructures narrowed to MILP, it can be reformulated into a single MILP. The main difficulty in this case are the common variables, for instance assume we have an inner MILP ${ }^{1}$ with variables $x, y$ and another inner MILP ${ }^{2}$ with variables $x, z$ (note that $x$ are the shared variables), then to integrate both of them into a single MILP we need to

- create the objective function $\min \left(d^{1} c_{x}^{1}+d^{2} c_{x}^{2}\right) x+c_{y}^{1} y+c_{z}^{2} z$, and
- create the constraints $A_{x}^{1} x+A_{y}^{1} y \leq /=/ \geq b^{1}$ and $A_{x}^{2} x+A_{z}^{2} z \leq /=/ \geq b^{2}$.
where
- $d^{1}$ and $d^{2}$ are the directions of MILP ${ }^{1}$ and MILP ${ }^{2}$, respectively;
- $c_{x}^{1}$ and ${ }_{c} x^{2}$ are the costs related with the shared variables of MILP ${ }^{1}$ and MILP ${ }^{2}$, respectively;
- $c_{y}^{1}$ and ${ }_{c} z^{2}$ are the costs related with the independent variables of MILP ${ }^{1}$ and MILP ${ }^{2}$, respectively;
- $A_{x}^{1}$ and $A_{x}^{2}$ are the LHS matrices related with the shared variables of MILP ${ }^{1}$ and MILP ${ }^{2}$, respectively;
- $A_{y}^{1}$ and $A_{z}^{2}$ are the LHS matrices related with the independent variables of MILP ${ }^{1}$ and MILP ${ }^{2}$, respectively; and
- $b^{1}$ and $b^{2}$ are the RHS vectors of MILP ${ }^{1}$ and MILP $^{2}$, respectively.

The diagram in Figure 7.3 is a representation of this composition.
By using this integration mechanism we can define the $\mathrm{ARR}^{\Sigma}$ to reformulate d_Composition_C(d_MILP_C, d_MILP_C) into d_MILP_C.

```
d_Composition_MILP_MILP_to_MILP_ARR : d_ARR_AIgebraic
[
    A -> d_Composition_C(d_MILP_C, d_MILP_C),
    B }->\mathrm{ d_MILP_C,
    indexA -> [_tiD(X,[p1\ldotsxi,p2\ldotsxi])=xii, -tiD(Y, [p1\ldotsxi])=yii, _tiD(Z, [p2\ldotsxi]=zii),
```



Figure 7.3: Composition of two MILP subproblems with shared variables

```
    _tiD(X,[p1\ldotsxr,p2\ldotsxr])=xir, _tiD(Y, [p1\ldotsxr])=yir, _tiD(Z, [p2\ldotsxr]=zir),
    p1..cons = j1, p2..cons = j2
indexB ->> [j,
dimRel }->\mathrm{ - [colsl = _tiD(X,[p1..xi,p2 ..xi]) + -tiD(Y, [p1..xi]) + -tiD(Z, [p2 ..xi]),
        colsR = -tiD(X,[p1\ldotsxr,p2\ldotsxr]) + -tiD(Y, [p1..xr]) + -tiD(Z, [p2\ldotsxr]],,
    cons = p1..cons + p2..cons
arg_map -> [
        B..ci = $([ $(A..p1..dir*A..p1..ci(xii)+A..p2...dir*A..p2..ci(xii), [xii]),
        $(A\ldotsp1\ldotsdir*A..p1..ci(yii), [yii]),
            $(A..p2\ldotsdir*A..p2\ldotsci(zii), [zii]) ]),
        B..cr = $([ $(A..p1..dir*A..p1..ci(xir)+A..p2..dir*A..p2..ci(xir), [xir]),
                $(A..p1..dir*A..p1...ci(yir), [yir]),
                $(A\ldotsp2\ldotsdir*A\ldotsp2\ldotsci(zir), [zir]) ]),
            B..Ai = $([ // LHS int matrix
                    [ $(A\ldotsp1..Ai(j1,xii), [j1,xii]), $(A..p1..Ai(j1,yii), [j1,yii]),
                                    $(0 , [j1,zii]) ],
                            [ $(A\ldotsp2\ldotsAi(j2,xii), [j2,xii]), $(0, [j2,yii]),
                    #(A\ldotsp2\ldotsAi(j2,zii),[j2,zii])]
            B..Ar = $([]// LHS real matrix
                    [ $(A..p1\ldotsAr(j1,xir), [j1,xir]), $(A..p1...Ar(j1,yir), [j1,yir]),
                        $(0
                            $(A\ldotsp2\ldotsAr(j2,zir), [j2,zir]) ]
            B_.rels=$([{ $(A\ldotsp1\ldotsrels(j1), [j1]), $(A\ldotsp2\ldotsrels(j2), [j2]) ]),
            .dir = 'min'
                xi = [x1=[-tiD(X, [p1\ldotsxi,p2\ldotsxi])], y1=[-tiD(Y, [p1\ldotsxi])], z1=[-tiD(Z, [p2\ldotsxi])]],
                xr = [x2=[-tiD(X, [p1\ldotsxr,p2\ldotsxr])], y2=[-tiD(Y, [p1\ldotsxr])], z2=[-tiD(Z, [p2\ldotsxr])]],
                xi(x1) = lower(_max([ -Iw(A..p1..xi(xii)), -Iw(A..p2\ldotsxi(xii)) ])),
                xi(x1) = upper(-min ([-Iw(A\ldotsp1\ldotsxi(xii)), -lw(A\ldotsp2\ldotsxi(xii)) ])),
                xi(y1) = lower(-lw(A..p1..xi(yii))),
                B.xi(y1) = upper(_up(A..p1...xi(yii))),
                xi(z1) = lower(_Iw(A..p2...xi(zii))),
                .xi(z1) = upper(-up(_si(A..p2\ldotsxi(z1)))),
                _.xr(x2) = lower(_max([ -Iw(A..p1\ldotsxr(xir)), -Iw(A..p2\ldotsxr(xir)) ])),
                xr(x2) = upper(_min([ -Iw(A..p1..xr(xir)), _lw(A..p2\ldotsxr(xir)) ])),
                .xr(y2) = lower(_lw(A..p1...xr(yir))),
            .xr(y2) = upper(-up(A..p1..xr(yir))),
            B..xr(z2) = lower(_lw(A..p2...xr(zir))),
            B..xr(z2) = upper((up(A..p2\ldotsxr(zir)))
            > [,
            A..p1..xi(xii) = B..xi(x1),
            A..p1..xi(yii) = B..xi(y1),
            A..p1\ldotsxr(xir) = B ..xr(x2),
            A..p1..xr(yir) = B..xr(y2),
            A..p2...xi(xii) = B..xi(x1),
            A..p2..xi(zii) = B..xi(z1),
            A\ldotsp2\ldotsxr(xir)}=\textrm{B}\ldotsxr(x2)
            A..p2\ldotsxr(zir) = B...xr(z2)
    ]
```

].

Other combinations of d_MILP_C and d_LP_C as substructures of d_Composition_C can conduct to similar $\mathrm{ARR}^{\Sigma}$ to treat those cases. We only have to be careful with the resulting structure, that it is always d_MILP_C except for the case when both substructures are d_LP_C (in that case the generated structure must be d_LP_C.

### 7.4 Applying the $\mathrm{ARR}^{\Sigma_{\mathrm{S}}}$ to HCP

Taking the HCP formulation we defined in $\S 7.2$, we could apply a combination of the previously defined ARR ${ }^{{ }_{S}}$ until finally obtain a MILP formulation. To show how the HCP formulation is modified by the application of the ARR ${ }^{\Sigma}$ we will use the HCP algebraic formulation combined with the graphical representation, pointing out the latest reformulation applied. To do so, we will dim all the model except for the structure being transformed, and the new structure obtained will have a gray background color (instead of white).

We start from the original HCP formulation.


First we apply the $\mathrm{ARR}^{\Sigma}$ d_VAbs_to_LP_oncond_OFMin_ARR (§7.3.3) to the structure vabsof in the $^{\text {( }}$ formulation, obtaining the following.


A new structure of class d_LP_C substitutes the structure vabsof, even if in the actual reformulated model vabsof is exchanged with the track structure _tr (vabsof, d_LP_C). To keep the example simple
we will only show the tail of the track structures.
We can now reformulate semicof by applying the $\operatorname{ARR}^{\sum}$ d_SemiContinuous_LP_SingleBV_to_MILP_ARR (cf. §7.3.4). Observe that semicof meets the criteria for this reformulation, since it has a substructure of class d_LP_C and another of class d_SingleBV_C.


Since d_IndComposition_C has a substructure of type d_MILP_C, then we can apply the ARR ${ }^{\sum}$ d_IndComposition_MILP_to_MILP_ARR (cf. §7.3.9). Notice that the MILP has the same free indices semicof had in the original model $(i \in M, j \in N)$, so this reformulation will integrate the $\|M\| *\|N\|$ replications of the inner MILP. Moreover, after doing this we can apply the ARR ${ }^{{ }^{D}}$ d_OFMin_MILP_to_MILP_ARR (cf. §7.3.8), since of has d_MILP_C has its inner structure.


Let us move to the constraints part, staring by reformulating sabsc using $\mathrm{ARR}^{\Sigma}$ d_SAbs_to_Composition_LP_ProdCC_ARR (cf. §7.3.2).


Although in this case the complexity of the model augmented a little bit, this will allow us to simplify it further by applying $\mathrm{ARR}^{\sum}$ d_ProdCC_to_MILP_ARR (cf. $\S 7.3 .5$ ) to the d_ProdCC_C structure class.


Observe that at this point the algebraic representation is in MILP form. However, the formulation still have to undergo the following reformulations to be completely transformed into a d_MILP_C.


Note how in this example the reformulations are applied only when the narrowing requisites are met. Only at that point the corresponding $\operatorname{ARR}^{\Sigma}$ can be applied to transform the structure.

Thanks the the deductive power of $\mathcal{F}$ LORA-2, the system easily detects which ARRs it can apply to a certain (maybe intermediate) formulation, allowing the creation of all possible reformulations.

## Chapter 8

## More focused structures and reformulations

Abstract $\qquad$
In this chapter we will focus on a set of structures for which there are specialized solution methods, and we will study how we can relate them by applying the appropriate reformulation rule. We will start from "simple" classes (maybe already defined throughout the previous chapters), and we will end with some non-linear structures like Quadratic Multicommodity Minimum Cost Flow (Fixed Charge) (QMMCF(FC)) and Second Order Cone Program (SOCP). Along this chapter we will build, step by step, a diagram representing the possible reformulations between the defined structures.

### 8.1 Rounding Up previously defined structures

Throughout the previous chapters we defined some structures that will be useful at this moment. The following list summarizes the name, description and reference in the text of those structures.

- d_IC_C - integrality constraint (cf. Listing 1.3),
- d_Linear_Constraints_C - linear constraints without objective function (cf. Listing 1.4),
- d_LP_C - linear problem (cf. Listing 1.5),
- d_MILP_C - mixed integer linear problem (cf. Listing 2.23),
- d_Simple_Selection_C - simple binary values selection based on a vector of costs (cf. Listing 1.12),
- d_MCF_C - minimum cost flow (cf. Listing 1.6),
- d_MMCF_FC_C - multicommodity minimum cost flow (fixed charge) (cf. Listing 5.4),
- d_B_MILP_C - block representation of a mixed integer linear problem (cf. Listing 1.9),
- d_B_Lagrangian_Relax_C - block representing a Lagrangian relaxation (cf. Listing 1.14).

In $\S 4.1 .2$ we mentioned the knapsack structures and the advantages of explicitly recognizing it. We can then define the following respective structure class to represent the continuous knapsack problems,

```
d_KnapSack_C :: d_LeafProblem_C
[
    dim_var -> [E],
    args -> [
        x = d_vector(d_var, [E]),
        c = d_vector(d_constant, [E]), // cost
        w = d_vector(d_constant, [E]), // weight
        m = d_constant // capacity
        ]
].
```

having the following LP formulation,

$$
\begin{aligned}
& \min \sum_{i \in \mathrm{E}} \mathrm{c}_{i} \mathrm{x}_{i} \\
& \text { s.t. } \\
& \qquad \sum_{i \in \mathrm{E}} \mathrm{w}_{i} \mathrm{x}_{i} \leq \mathrm{m} \\
& \quad 0 \leq \mathrm{x}_{i} \leq b_{i} \quad i \in \mathrm{E}
\end{aligned}
$$

The upper bounds $b_{i}$ are not defined as a parameter of the structure, because they will belong to the variable definition (cf. §2.2.1).

### 8.1.1 Reformulations

Starting with these structure classes there are some ARRs that could be defined.

## MILP to block MILP and a more general case

The MILP LfP can be easily reformulated into a block MILP, using the following ARR $^{\Sigma}$,
Listing 8.2: MILP to block MILP

```
d_MILP_to_B_MILP_ARR : d_ARR_Algebraic
[
    A \(\rightarrow\) d_MILP_C
    B \(\rightarrow\) d_B_MILP_C(? - ? -\()\),
    indexA \(\rightarrow\) [colsl \(=\mathrm{i}\), cols \(R=r\), cons \(=j]\),
    indexB \(\rightarrow\) [],
    \(\operatorname{dimRel} \rightarrow[|p \ldots c o n s=c o n s,|p \ldots c o l s=c o l s|+c o l s R\), ic...d=cols|],
    arg_map \(\rightarrow\) [ \(\ldots\)..Ip..A \(=\$([\)
                                    \$(A..Ar \((j, r), \quad[j, r])\),
                                    \(\$(A \ldots A i(j, i),[j, i])\)
                                    ]
            \(\left.B \ldots I p \ldots c=\$()^{\prime}\right)^{\prime}\),
                        \$(A..cr(r), [r]),
                            ]),
        B..Ip..b \(=A \ldots\). \({ }^{\prime}\)
        B..lp..rels = A..rels
        B..Ip..dir =A..dir
        B..|p..x = [real=A..colsR, int=A..colsl],
        B..Ip...x(real) \(=\) lower(_Iw(A..xr)),
        B..|p...x(real) \(=\operatorname{upper}(\ldots \mathrm{up}(\mathrm{A} \ldots \times r))\),
        B..Ip...x(int) \(=\operatorname{lower}(\operatorname{Im}(A \ldots x i))\),
        B..Ip...x(int) \(=\operatorname{upper}(\operatorname{lup}(A \ldots x i))\)
    ans_map \(\rightarrow\) [,
            \(A \ldots x r=B \ldots \mid p \ldots x(\) real \()\),
            \(A \ldots x i=B \ldots I p \ldots x(i n t)\)
            ],
    fixTl \(\rightarrow\) [
        \(X=B \ldots \mid p \ldots x(\) real \()\),
            \(X=B \ldots \mid p \ldots x(\) real \()\)
\(Y=B \ldots l p \ldots x(\) int \()\)
```

```
].
```

Note that the application of d_MILP_to_B_MILP_ARR allow us to use solvers (for d_B_MILP_C) that depends on sub-solvers assigned to the inner d_LP_C. Morevover, we could transitively reformulate d_LP_C into any of its formats (i.e. d_LP_MPS_C). So starting from a m:d_MILP_C, we could apply d_MILP_to_B_MILP_ARR and obtain the track structure _tr (m, d_B_MILP(d_LP_C, d_IC_C)). Then, we could select d_LP_C and transform it into d_LP_MPS_C, obtaining the track strcuture _tr (m, d_B_MILP(_ts(d_LP_C, d_LP_MPS_C), d_IC_C)).

Furthermore, d_B_MILP_C structure can be seen as a particular case of a Branch and Bound structure class, that can be defined as follows,

Listing 8.3: Branch and Bound block

```
d_B_Branch_Bound_C :: d_Block_C [
    ids }->\mathrm{ [ sub, ic ],
    subsC }->\mathrm{ [ d _BB_C, d_loc(d_IC_C)],
    link > [([X,Y],d_all), ([X],d_all)]
].
d_BB_C :: d_Component_C [abstract]. // Auxiliary structure class
```

This structure allows the application of general Branch and Bound solvers, that will depend on the result of the sub-solver assigned to sub.

We could then make the ARR between d_MILP_C and d_B_Branch_Bound_C where the inner structure is d_LP_C,

Listing 8.4: MILP to Branch and bound with LP

```
d_MILP_to_BB_LP_ARR : d_ARR_AIgebraic
[
    A -> d_MILP_C,
    B -> d_B_Branch_Bound_C(d_LP_C, ?_),
    indexA -> [colsl=i,colsR=r,cons=j],
    indexB -> [],
    dimRel - [sub..cons=cons, sub..cols=colsl+colsR, ic..d = colsl],
    arg_map -> [ B..sub..A = $([
        #(A.. Ar(j,r), [j,r]),
            $(A..Ai(j, i), [j,i])
            B..sub ..c c = $),
                        $(A\ldotscr(r), [r]),
                    $(A..ci(i), [i])
                ]),
            B..sub..b = A..b,
            B..sub..rels = A..rels,
            B..sub..dir = A..dir
            B..sub..x = [real=A..colsR, int=A..colsl],
            B..sub..x(real) = lower(_Iw(A..xr)),
            B..sub..x(real) = upper(_up(A..xr)),
            B..sub\ldotsx(int) = lower(_Iw(A..xi)),
            B..sub\ldotsx(int) = upper(-up(A..xi)),
    ans_map -> [
        A..xr = B ..sub .. x(real) ,
            A..xi = B..sub..x(int)
    fixTI M [ ],
        X = B..sub .. x(real)
        Y = B..sub .. . (int)
            ]
].
```

Note that the same principle of transitive ARR usage can be apply in this case, making reformulation of the d_B_Branch_Bound_C inner structure.

## MMCF(FC) to MILP

As could be seen in $\S 1.3$ the $\mathrm{MMCF}(\mathrm{FC})$ has the following a MILP representation.

$$
\begin{align*}
& \min \sum_{k \in K} \sum_{(i, j) \in E} \mathrm{c}_{i j}^{k} * \mathrm{flow}_{i j}^{k}+\sum_{(i, j) \in E} \mathrm{f}_{i j} \text { desg }_{i j} \\
& \text { s.t. } \\
& \sum_{j \in N^{+}(i)} \text { flow }_{i j}^{k}-\sum_{j \in N^{-}(i)} \text { flow }_{j i}^{k}=S D_{i}^{k} \quad i \in N, k \in K \\
& 0 \leq \text { flow }_{i j}^{k} \leq \mathrm{b}_{i j}^{k} \quad(i, j) \in E, k \in K \\
& \sum_{k \in K} \text { flow }_{i j}^{k} \leq \mathrm{u}_{i j} \quad(i, j) \in E \\
& \sum_{k \in K} \text { flow }_{i j}^{k} \leq \mathrm{u}_{i j} \text { desg }_{i j} \quad(i, j) \in E  \tag{8.1.1}\\
& 0 \leq \operatorname{desg}_{i j} \leq 1 \quad(i, j) \in E \\
& \operatorname{desg}_{i j} \text { integer } \quad(i, j) \in E
\end{align*}
$$

Note that from this representation we can see that the flow variables are constraint to be semicontinuous (SC) by (8.1.1). The fact of recognizing the SC constraints in a MILP will help us in the definition of of further reformulations. To do so, we will create an structure d_MILP_SC_C that explicitly recognizes the SC constraint information.

## Listing 8.5: Semi-Continuous MILP

```
d_MILP_SC_C :: d_LeafProblem_C
[
    // Dimension description
    // cons --> linear constraints
    // colsR -> columns for real variables
    // colsl -- columns for int variables
    // colsR = colsl*fact
    // The structure forces xi to be binary
    // and xr to have lower bound 0
    dim_var -> [cons, colsR, colsl],
    args -> [
            // OF parameters (int part)
            ci -> d_vector(d_constant, [colsl]),
            // OF parameters (real part)
            cr -> d_vector(d_constant, [colsl]),
            // Semicontinuous constraints parameter
            Asc -> d_vector(d_constant, [colsl, colsR]),
            bsc -> d_vector(d_constant, [colsl]),
            // Linear constraint parameters
            Ai -> d_vector(d_constant, [cons, colsl]),
            Ar -> d_vector(d_constant, [cons, colsR]),
            rhs -> d_vector(d_constant, [cons]),
            rels -> d_vector(d_constant, [cons]),
            fact -> d_constant
            //Variables
            xr -> d_vector(d_var, [colsR]),
            xi -> d_vector(d_var, [colsl]),
            ]
].
```

We can easily define an $\mathrm{ARR}^{\sum}$ from d_MILP_SC_C to d_MILP_C.
Listing 8.6: MILP-SC to MILP

```
d_MILP_SC_to_MILP_ARR : d_ARR_Algebraic
[
    A -> d_MILP_SC_C.
```

```
B \(\rightarrow\) d_MILP_C
indexA \(\rightarrow\) [colsl=(ii, ii1), colsR=ir, cons=j]
indexB \(\rightarrow\) []
dimRel \(\rightarrow\) [cons=cons+colsl, colsl=colsl, colsR=colsR],
arg_map \(\rightarrow\) [ B cr \(=A\)
    \(\mathrm{B} \ldots \mathrm{cr}=\mathrm{A} \ldots \mathrm{Cr}\),
\(\mathrm{B} \ldots \mathrm{ci}=\mathrm{A} \ldots \mathrm{ci}\)
    \(\mathrm{B} . . \mathrm{Ai}=\$([/ / l e f t\) hand matrix (for int variables)
                            \$(A..Ai(j, ii), [j, ii]),
                            \(\$(\ldots \operatorname{cs}([A \ldots b i(\mathrm{ii}, \mathrm{i} 11)->(\mathrm{ii}=\mathrm{i} i 1), 0]), \quad[\mathrm{ii}, \mathrm{ii} 1])\)
        B. Ar \(=\$\left(\left[{ }^{\text {] }}\right.\right.\)
            ]),
        B..Ar \(=\) ([ //left hand matrix (for integer variables)
            // flow constraints
                \$(A...Ar(j, ir), \([j, i r])\),
                \$(A..Asc(ii, ir), [ii, ir \(]\) )
            \(\mathrm{B} \ldots \mathrm{b}=\$\left(\left[\begin{array}{l}\mathrm{l})^{\prime} \text {, right hand vector } \\ \$(\mathrm{~A}\end{array}\right.\right.\)
                    \(\$(A \ldots r h s(j),[j])\),
\(\$(0\),
            \$(0
        B...rels \(=\$^{\prime}\left(\left[\begin{array}{l}\text { // relation vector } \\ \$(\mathrm{~A} . \mathrm{rels}(\mathrm{j}), \\ \$(\mathrm{j}]),\end{array}\right.\right.\)
                        \(\$('=<',[\mathrm{ii}])\)
                ]),
    B..dir = A..dir
    B.. xr \(=\) lower (0)
    B..xr \(=\operatorname{upper}(-\operatorname{up}(A . . x r))\),
    B...xi \(=\operatorname{lower}(0)\),
    B...xi \(=\) upper (1)
    ans_map \(\rightarrow\) [
        \(A \ldots x r=B \ldots x r\),
        \(A \ldots \times i=B \ldots \times i^{\prime}\)
    ]
```

].

Now we can build an $\mathrm{ARR}^{\sum}$ to reformulate d_MMCF_FC_C into d_MILP_SC_C.
Listing 8.7: MMCF(FC) to MILP-SC

```
d_MMCF_FC_to_MILP_SC_ARR : d_ARR_Algebraic
[
    A -> d_MMCF_FC_C,
    B }->\mathrm{ d_MILP_C,
    indexA->[E=(e,e1),K=(k,k1) ,N=n],
    indexB -> [],
    dimRel }->[\mathrm{ cons =N*K+E, colsl=E,colsR=E*K],
    arg_map l> [
                B..cr = A..c,
                B..ci = A..f
            B..Ar = $([ //left hand matrix (for real variables)
                    [ // flow constraints
                    $([1->(A..SN(e)=n, k=k1),
                            -1->(A..EN(e)=n, k=k1)
                            0]
                            [(n,k1), (e,k)])
                            ],
                            [ // mutual arc capacity constraints
                            $([1-> (e=e1), 0], [e1, (e,k)])
            B..Ai = $([] [//
            [ /lleft hand matrix (for integer variables)
            [// flow constraints
                    $(0, [(n,k1), e])
                            ],
                            // mutual arc capacity constraints
                    $(0, [e1, e])
            B..rhs = [)],
                        $(A..SD(k,n), [(n,k)]),
            $(A..u(e),, [e] )
            B...rels = $([['// relation vector
                    $('=',}[(\textrm{n},\textrm{k})])
```

```
*
```

```
    B..Asc = $([1->>(e=e1), 0], [e1, (e,k)])
```

    B..Asc = $([1->>(e=e1), 0], [e1, (e,k)])
    B..bsc = $(A..u(e), [e]),
    B..bsc = $(A..u(e), [e]),
    B..fact = _card(A..K),
    B..fact = _card(A..K),
    B..dir = 'min
    B..dir = 'min
    B..xr = upper(A..b)
    B..xr = upper(A..b)
    ans_map -> [,
ans_map -> [,
A..flow = B..xr,
A..flow = B..xr,
A..desg = B...xi
A..desg = B...xi
].

```

Note that when we transform d_MMCF_FC_C into d_MILP_SC_C, by transitivity we are transforming it also into d_MILP_C (applying d_MMCF_FC_to_MILP_SC_ARR) and then d_MILP_SC_to_MILP_ARR).

\section*{MMCF(FC) to Lagrangian Relaxations}

The d_MMCF_FC_C structure can also be reformulated into a flow Lagrangian relaxation, as exposed in \(\S 5.2 .6\). However this is not the only relaxation that can be done. Let us see again the MILP representation of MMCF (FC).
\[
\begin{align*}
& \min \sum_{k \in K} \sum_{(i, j) \in \mathrm{E}} \mathrm{c}_{i j}^{k} * \mathrm{flow}_{i j}^{k}+\sum_{(i, j) \in \mathrm{E}} \mathrm{f}_{i j} \mathrm{desg}_{i j} \\
& \text { s.t. } \\
& \sum_{j \in N^{+}(i)} \text { flow }_{i j}^{k}-\sum_{j \in N^{-}(i)} \text { flow }_{j i}^{k}=S D_{i}^{k} \quad i \in \mathrm{~N}, k \in \mathrm{~K} \quad\left(\nu_{i}^{k}\right)  \tag{8.1.2}\\
& 0 \leq \text { flow }_{i j}^{k} \leq \mathrm{b}_{i j}^{k} \quad(i, j) \in \mathrm{E}, k \in \mathrm{~K}  \tag{8.1.3}\\
&  \tag{8.1.4}\\
& \sum_{k \in K} \operatorname{flow}_{i j}^{k} \leq \mathrm{u}_{i j} \quad(i, j) \in \mathrm{E}  \tag{8.1.5}\\
& \sum_{k \in K} \operatorname{flow}_{i j}^{k} \leq \mathrm{u}_{i j} \text { desg }_{i j} \quad(i, j) \in \mathrm{E}  \tag{8.1.6}\\
& \text { flow }_{i j}^{k} \leq \mathrm{b}_{i j}^{k} \text { desg }_{i j} \quad(i, j) \in \mathrm{E}, k \in \mathrm{~K}  \tag{8.1.7}\\
& 0 \leq \operatorname{desg}_{i j} \leq 1 \quad(i, j) \in \mathrm{E}  \tag{8.1.8}\\
& \operatorname{desg}_{i j} \operatorname{integer} \quad(i, j) \in \mathrm{E}
\end{align*}
\]

We could obtain a Knapsack relaxation by dualizing the flow conservation constraints (8.1.2). The resulting Lagrangian dual is:
\[
L R=\max _{\nu}\left\{\sum_{k \in \mathrm{~K}} \sum_{i \in \mathrm{~N}} \nu_{i}^{k} \mathrm{SD}_{i}^{k}+\min _{(\text {flow,desg }) \in(F, D)} \sum_{k \in \mathrm{~K}} \sum_{(i, j) \in \mathbf{E}}\left(\mathrm{c}_{i j}^{k}+\nu_{j}^{k}-\nu_{i}^{k}\right) \mathrm{flow}_{i j}^{k}+\sum_{(i, j) \in \mathbf{E}} \mathrm{f}_{i j} \operatorname{desg}_{i j}\right\}
\]
where the set \((F, D)\) is defined by constraints (8.1.3) to (8.1.8). After solving \(|\mathrm{E}|\) continuous knapsack problems:
\[
\begin{aligned}
& g_{i j}=\min \sum k k \in \mathrm{~K}\left(\mathrm{c}_{i j}^{k}+\nu_{j}^{k}-\nu_{i}^{k}\right) \mathrm{flow}_{i j}^{k} \\
& \\
& \sum_{k \in \mathrm{~K}} \mathrm{flow}_{i j}^{k} \leq \mathrm{u}_{i j} \\
& 0 \leq \operatorname{flow}_{i j}^{k} \leq \mathrm{b}_{i j}^{k} \quad k \in \mathrm{~K},
\end{aligned}
\]
the Lagrangian subproblem can be solved by inspection as:
\[
\begin{equation*}
\min _{\operatorname{des} \in\{0,1\} \mid} \sum_{(i, j) \in \mathbf{E}}\left(\mathrm{f}_{i j}+g_{i j}\right) \operatorname{desg}_{i j} \tag{8.1.9}
\end{equation*}
\]

To create the ARR to represent this process we will need a block that will represent the mentioned sub-problem. This block will be called Independent Replication Selection, and it will solve a set of substructures to compute the \(g\) and then perform a simple selection over a set of binary variables.

Listing 8.8: Independent Replication Selection block
```

d_B_Ind_Replication_Selection_C :: d_Block_C [
ids -> [ sub, sel],
subsC -> [ d_IRS_C, d_loc(d_Simple_Selection_C )],
link -> [([X],d_all),
rpIR -> [sub = Y, sel = 1]
].
d_IRS_C :: d_Component_C [abstract]. // Auxiliary structure class

```

Note that the amount of binary variables must be equal to the amount of replications of the inner sub-structure (e.i. \|sub. freeinds \(\|\) ).

Using this structure class we can define the following \(\mathrm{ARR}^{\Sigma}\),
Listing 8.9: MMCF(FC) to Lagrangian Knapsack Relaxation
```

d_MMCF_FC_to_Lagrangian_KS_Relax_ARR : d_ARR_AIgebraic
d_M
A -> d_MMCF_FC_C,
B ->> d_Lagrangian_Relax_C(d_B_Ind_Replication_Selection_C(d_KnapSack_C, ?_), ?_),
indexA -> [E=e,K=(k,k1),N=n],
indexB }->\mathrm{ [],
dimRel }->\mathrm{ [b1..sub..E=K, b1..sel..d1=E, linking..d1=K*E, linking..d2=N*K],
arg_map -> [
//___ Knapsack part
// Note that e is left untouched. Therefore e will be
// the free-index on which the Knapsack structure will be replicated.
B..sub..sub..c = \$(A..c(k,e), [k]),
B..sub..sub...w = \$(1, [k]),
B..sub..sub..m = A..u,
//- Simple Selection part
B..sub..sel..f = A..f,
//-_Linear Constraints part (linking constraints)
B..linking..A = \$([1-> (A..SN(e)=n, k=k1),
-1->(A..EN(e)=n, k=k1),
0],
[(n,k1), (e,k)]),
B..linking...rels = \$('=', [(n,k)]),
B..linking..b = \$(A..SD(k,n), [(k,n)])
ans_map -> ['
A..flow = B..sub..sub... . .
A..desg = B..sub..sel..y
fixTl > [,
X = (B..sub..sub .. x, B..sub..sel..y),
sub.. X = B..sub..sub...
sub..Y = B..sub..sel..y
]
].

```

\subsection*{8.1.2 Reformulation Diagram}

Using the structures defined so far and the reformulations between them we can depict a diagram (see Figure 8.1). In this diagram we include the d_MCF_to_LP_ARR defined in 5.2.6. On the other hand, we did not include the possible reformulations involving the different formats. For instance, considering
- MPS - for LP and MILP;
- DIMACS - for MCF [3];
- Canad - for MMCF(FC) [13],


Figure 8.1: Initial set of ARRs and structures
we could easily define a set of \(\mathrm{ARR}^{\Sigma_{S}}\) (in both ways) to reformulate to (from) the specific formats. These reformulation are generally a trivial mapping of the structure's parameters. Therefore we will assume for simplicity that \(\operatorname{ARR}^{\Sigma_{\mathrm{s}}}\), between the formats and the global structure class, always exist.

\subsection*{8.2 Some Convex Structures}

In this section we will study a particular case of nonlinear structure,
\[
\begin{aligned}
& \min \sum_{i \in I} f_{i}\left(p_{i}\right)+c_{i} u_{i} \\
& \quad A_{i} p_{i} \leq b_{i} u_{i} \quad i \in I \\
& \quad(p, u) \in \mathcal{O} \\
& u \in\{0,1\}^{|I|} \\
& \quad p \in \mathbb{R}^{m_{i}}, i \in I
\end{aligned}
\]
where \(f_{i}\) is a non-linear function over continuous variables and \(\mathcal{O} \subseteq \mathbb{R}^{|I|+\sum_{i \in I} m_{i}}\) representing a set of linear constraints over \(p_{i}\). If we consider \(f_{i}\) to be a quadratic function \(\left(a p_{i}^{2}+d p_{i}\right)\) then the structure we will obtain is a Mixed Integer Quadratic Program with Semi-continuous variables and Separable Objective function (MIQP-SC-SEP). Note that this structure can be seen as a composition (7.1.1) of d_MILP_SC_C and a Quadratic Function ( qF ) structure,
\[
\min \sum_{i \in \mathrm{D}}\left(\mathrm{q}_{i} x_{i}^{2}+\mathrm{c}_{i} x_{i}\right)
\]

We could create the following structure class to represent a qF ,
Listing 8.10: Quadratic Function
```

d_Quadratic_F_C :: d_LeafProblem_C
[
dim_var -> [D],

```
```

args -> [
x = d_vector(d_var, [D]),
q = d_vector(d_constant, [D]),
c = d_vector(d_constant, [D])
]
].

```

The narrowing d_Composition_C(d_MILP_SC_C, d_Quadratic_F_C) (that represents the MIQP-SC-SEP structure) can be tackled with CPLEX as a general mixed integer quadratic program. However, recognizing this structure will allow us to use certain relaxations that will use Mixed Integer Second Order Cone Program and Semi-Infinite Perspective Cuts, defined afterwards.

In this section we will define two structure classes based on second-order cone program (SOCP) [41]. A second-order cone program (SOCP) is a convex optimization problem of the form,
```

$\min f^{T} x$
s.t.
$\left\|A_{i} x+b_{i}\right\|_{2} \leq c_{i}^{T} x+d_{i}, i \in[1 . . m]$
$F x=g$

```
where \(f \in \mathbb{R}^{n}, A_{i} \in \mathbb{R}^{n_{i} \times n}, b_{i} \in \mathbb{R}^{n_{i}}, c_{i} \in \mathbb{R}^{n}, d_{i} \in \mathbb{R}, F \in \mathbb{R}^{p \times n}\) and \(g \in \mathbb{R}^{p}\). The optimization variables are \(x \in \mathbb{R}^{n}\).

Based on this definition we can construct a structure class to represent the SOCPs.
Listing 8.11: Second Order Cone Program
```

d_SOCP_C :: d_LeafProblem_C
[
// Dimension description
// consL --> linear constraints
// Cineq --> conic inequalities
// Clrows-> rows in each conic inequality
// cols --> columns for real variables
dim_var -> [consL, consC, cols],
args -> [
// Objective function input
f = d_vector(d_constant, [cols]),
// linear constraints input
AL = d_vector(d_constant, [consL, cols]),
rhsL = d_vector(d_constant, [consL]),
relsL= d_vector(d_rel, [consL])
// conic inequalities
ACl = d_vector(d_constant, [Cineq, Clrows, cols]),
bCl = d_vector(d_constant, [Cineq, CIrows]),
dCI = d_vector(d_constant, [Cienq]),
cCl = d_vector(d_constant, [Cineq, cols]),
// Variables
x = d_vector(d_var, [cols]),
]
].

```

There are several solvers (mostly based on the interior point method) that tackle the SOCP structure. For instance, MOSEK [9], CPLEX [7], LOQO [8], CSDP [2], SDPA [11], and others.

There is an extension that can be done to the SOCP class, by specifying a subset of the variables as integer. This extension is called Mixed Integer \(S O C P\) and can be represented by the following structure class,

Listing 8.12: Mixed-Integer Second Order Cone Program
```

d_MISOCP_C :: d_LeafProblem_C
[
// Dimension description
// consL —> linear constraints
// Cineq -> conic inequalities

```
```

// Clrows }\longrightarrow\mathrm{ rows in each conic inequality
// colsl -> columns for integer variables
// colsR —> columns for real variables
// colsR = fact*colsl
dim_var -> [consL, Cineq, ClRows, colsl, colsR],
args -> [
// Objective function input
fi = d_vector(d_constant, [colsl]),
fr = d_vector(d_constant, [colsR]),
// linear constraints input
ALi = d_vector(d_constant, [consL, colsl]),
ALr = d_vector(d_constant, [consL, colsR]),
rhsL = d_vector(d_constant, [consL]),
relsL= d_vector(d_rel, [consL]),
// conic inequalities
ACli = d_vector(d_constant, [Cineq, Clrows, colsl]),
AClr = d_vector(d_constant, [Cineq, Clrows, colsR]),
bCl = d_vector(d_constant, [Cineq, Clrows]),
dCl = d_vector(d_constant, [Cineq]),
cCli = d_vector(d_constant, [Cineq, colsl]),
cClr = d_vector(d_constant, [Cineq, colsR]),
// column relation factor
fact = d_constant ,
// Variables
x = d_vector(d_var, [colsl]),
u = d_vector(d_var, [colsR])
]
].

```

In this case solution methods based on lift-and-project relaxation [146] or branch and bound outer approximations [40, 135] are available for solving MISOCP problems.

\subsection*{8.2.1 Reformulations}

\section*{MISOCP to Branch and Bound with SOCP}

To increase the amount of solution possibilities to the MISOCP structure class, we propose a reformulation into the general d_B_Branch_Bound_C structure class. This reformulation rule will split the MISOCP into a SOCP plus the integrality constraints.

Listing 8.13: MISOCP to Branch and bound with SOCP
```

d_MISOCP_to_BB_SOCP_ARR : d_ARR_AIgebraic
[
A -> d_MISOCP_LC_C
B -> d_B_Branch_Bound_C(d_SOCP_LC_C, ?_),
indexA -> [colsl=i,colsR=r,consL=j, Cineq=q, CIrows=w],
indexB -> [],
dimRel -> [sub..consL=consL, sub..cols=colsl+colsR,
sub..Cineq=Cienq, sub..Clrows=Clrows, ic..d = colsl],
arg_map -> [
B...sub.. AL = \$(
\$(A..ALr(j,r), [j, r]),
\$(A..ALi(j,i), [j,i])
B..sub ..f = = \$)}\mp@subsup{}{(}{([
\$(A..frr(r), [r]),
\$(A..fi(i), [i])
]),
B..sub..rhsL= = ..rhsL,
B..sub..relsL = A..relsL
B..sub..ACI = \$([
[ [[
\$(A..ACIr(q,w,r), [q,w,r]),
\$(A.. ACli(q,w,i), [q,w,i])
]
]),

```
```

        B..sub..bCI = A..bCI,
        B..sub..dCl = A..dCl,
        B..sub..cCl = $([
                        $(A..cClr(q,r), [q,r]),
                        $(A..cCli(q,i), [q,i])
            ]
        B..sub .. x = [r, ',
        B..ic..ivar = lower(0),
        B..ic..ivar = upper(1)
    ans_map -> [,
        A..xr = B ..sub ..x(real),
        A..xi = B..sub..x(int)
    fixTl -> [ ]
        X = B..sub .. x(real)
        Y = B..sub .. x(int)
    ]
    1. 
```

\section*{Perspective Relaxations}

Let us revisit the MIQP-SC-SEP algebraic representation,
\[
\begin{align*}
& \min \sum_{i \in I} f_{i}\left(p_{i}\right)+c_{i} u_{i}  \tag{8.2.1}\\
& \quad A_{i} p_{i} \leq b_{i} u_{i} \quad i \in I  \tag{8.2.2}\\
& \quad(p, u) \in \mathcal{O}  \tag{8.2.3}\\
& \quad u \in\{0,1\}^{|I|}  \tag{8.2.4}\\
& \quad p \in \mathbb{R}^{m_{i}}, i \in I \tag{8.2.5}
\end{align*}
\]
where \(f_{i}\) is a quadratic function and \(\mathcal{O} \subseteq \mathbb{R}^{|I|+\sum_{i \in I} m_{i}}\) representing a set of linear constraints. From this MINLP formulation we could define Perspective reformulation (PR) [52, 75], in the following way,
\[
\begin{align*}
& \min \sum_{i \in I} u_{i} f_{i}\left(\frac{p_{i}}{u_{i}}\right)+c_{i} u_{i}  \tag{8.2.6}\\
& \quad \text { (8.2.2), } \tag{8.2.7}
\end{align*}
\]

This reformulation provides stronger bounds than the continuous relaxation of the MINLP formulation \([75,20,71]\). Nevertheless, PR has a high nonlinearity in the objective function, due to the fractional term. Two reformulations have been proposed to overcome this problem: the first as a Mixed Integer SOCP [20, 149] and the second as a Semi-Infinite Linear Program [75].

PR can be written as a Mixed Integer SOCP (in the quadratic case) very simply, since when \(u_{i}>0\) a constraint \(t_{i} \geq a_{i} p_{i}^{2} / u_{i}\) can be algebraically transformed in \(\left(t_{i}+u_{i}\right)^{2} / 4 \geq a_{i} p_{i}^{2}+\left(t_{i}-u_{i}\right)^{2} / 4\), conducting to the following Mixed Integer SOCP,
\[
\begin{array}{rl}
\min & \sum_{i \in I} t_{i}+d_{i} p_{i}+c_{i} u_{i} \\
& \quad \sqrt{a_{i} p_{i}^{2}+\left(t_{i}-u_{i}\right)^{2} / 4} \leq\left(t_{i}+u_{i}\right) / 2 \quad i \in I \\
\quad & (8.2 .2), \quad(8.2 .3), \quad(8.2 .4), \quad(8.2 .5) \\
\quad t & t \mathbb{R}_{+}^{|I|}
\end{array}
\]

Now, from this last reformulation we could create an \(\operatorname{ARR}^{\Sigma}\) to transform d_Composition_C(d_MILP_SC_C, d_Quadratic_OF_C) into a d_MISOCP_C.
```

d_MIQP_SC_SEP_to_MISOCP_ARR : d_ARR_AIgebraic
[
A $\quad \rightarrow$ d_Composition_C(d_MILP_SC_C, d_Quadratic_F_C)
B $\rightarrow$ d_MISOCP_C,
indexA $\rightarrow$ [p1..colsl=(ii,i1),p1..colsR=ir, $1=i, p 1 \ldots$ cons=j],
indexB $\rightarrow$ [],
dimRel $\rightarrow$ [consL=p1..cons+p1..colsl, colsl=p1..colsl,
colsR=p1..colsR+p1..colsi, Cineq=p1..colsi, Clrows=2],
arg_map $\rightarrow$ [
B..fi $=A \ldots p 1 \ldots \mathrm{ci} * \mathrm{~A} \ldots \mathrm{p} 1 \ldots \mathrm{dir}$,
$\mathrm{B} . . \mathrm{fr}=\$([/ / \operatorname{colsR} * 2$

```

```

            ]) \({ }^{\$(1}\)
            \(\mathrm{B} . \mathrm{ALi}=\$\left(\left[{ }^{]} / /\right.\right.\)constcolsl \(\times\)cols
                    [\$(A..p1..Ai(j, ii), [j, ii])],
                    \([\$(-\operatorname{cs}([-1 * \mathrm{~A} \ldots \mathrm{p} 1 \ldots \mathrm{bsc}(\mathrm{ii})->(\mathrm{ii}=\mathrm{ii} 1), 0]), \quad[\mathrm{ii} 1, \mathrm{ii}])]\)
                    ]),
            B..ALr \(=\$([\) // cons+colsl \(\times\) colsR+colsl
                        [ // cons x cols \(R+\) colsl
                    \$(A..p1..Ar(j,ir), [j,ir]), \$(0, [j,ii])
                    ],
                        // colsl x colsR+colsl
                            \(\$(A \ldots p 1 \ldots A s c(i i, i r),[i i, i r]), \$(0,[i i, i i 1])\)
    ```



```

            \(\mathrm{B} . \mathrm{ACli}=\$\left(\left[\begin{array}{l}{\left[\begin{array}{l}\$ \\ / / \\ {[ } \\ {[/ / \text { colsl } \times 2 \times \mathrm{cols} \times 1 \times \mathrm{colsl}}\end{array}\right]}\end{array}\right.\right.\)
                    \(\$(-\operatorname{cs}([-0.5->(\mathrm{ii}=\mathrm{ii} 1), 0]), \quad[\mathrm{ii}, \mathrm{i}, \mathrm{ii} 1])\)
                        [ \({ }_{\$} / 1 /\) colsl \(\times 1 \times\) cols
                    \(\$(0, \quad[\mathrm{ii}, \mathrm{i}, \mathrm{ii} 1])\)
    ```

```

                                    [ \(/ /(0\), colsl \(\times 1 \times\) colsR+colsl \([\mathrm{ii}, \mathrm{i}, \mathrm{ir}])\),
                                    \(\$(-\operatorname{cs}([0.5->(\mathrm{i}=\mathrm{ii} 1), 0]),[\mathrm{ii}, \mathrm{i}, \mathrm{ii} 1])\)
                                    [' // colsl \(\times 1 \times\) cols \(R+\) colsl
                                    \(\$\left(\ldots \operatorname{cs}\left(\left[\mathrm{~A} . . \mathrm{p} 2 \ldots \mathrm{q}(\mathrm{ir})^{\wedge} 0.5->(\mathrm{ir}>=\mathrm{A} . . \mathrm{p} 1 \ldots \mathrm{fact} * \mathrm{i}\right.\right.\right.\)
                                    (ir<(Ap..p1..fact+1)*ii), 0]),
                                    \(\$\left(0,\left[\begin{array}{l}[i i, i, i r]), \\ [i, i i 1])\end{array}\right.\right.\)
                                    ]) \({ }^{\text {] }}\)
            \(\mathrm{B} . . \mathrm{bCl}=0\),
            \(\mathrm{B} . \mathrm{dCl}=0\),
            B...cCli \(=\$(-\operatorname{cs}([0.5->(i i=i i 1), 0]), \quad[i i, i i 1])\),
            B..cCIr \(=\$([\)
    ```

```

                                    ]) \({ }^{\text {] }}\)
            B..fact \(=A . . f a c t+1\),
            B..u = lower (0),
            B..u \(=\) upper (1),
            B.. \(x=[p=[p 1 \ldots\) colsR \(], t=[p 1 \ldots\) colsl \(]]\),
            B..x \(=\) lower (0),
            \(B \ldots x(p)=\operatorname{upper}(\operatorname{up}(A \ldots p 1 \ldots x r))\)
    ans_map \(\rightarrow \stackrel{\text { [, }}{ }\)
            A..p1...xi \(=B \ldots x(p)\),
            \(A \ldots p 1 \ldots x r=B \ldots u\)
        ]
    ```

Note in this case we used the number 1 as a dMV for A, meaning that it represents a dimension with one element. The index declared for 1 , using \(1=\mathrm{i}\), can take the values in the set \(\{0\}\). Furthermore, observe that in the ans_map we assigned B... \(\times(\mathrm{p})\) to \(\mathrm{A} . . \times\), ignoring the \(t\) variables that are also part of B...x. This can be done due to the variable partitioning we previously \(\operatorname{did}(B . . x=[p=[c o l s R], t=[\) colsl \(]])\), in which we explicitly identify the variables \(p\) and \(t\).

On the other hand, an alternative formulation of PR [75] can be done by representing \(u f(p / u)+\) \(c u\) with the following infinite family of linear inequalities, called perspective cuts,
\[
v \geq s p+(c+f(\bar{p})-s \bar{p}) u
\]
indexed over \(\bar{p} \in \cup_{i \in I} \mathcal{P}_{i}\), with \(\mathcal{P}_{i}=\left\{p_{i} \mid A_{i} p_{i} \leq b_{i}\right\}\) having that \(\left\{p_{i} \mid A_{i} p_{i} \leq 0\right\}=\{0\}\), and \(s \in \partial f(\bar{p})\), where \(\partial f(\bar{p})\) is the sub-differential of \(f\) at \(\bar{p}\). Since we assumed that \(f\) is quadratic, the we can describe the following Semi-Infinite MINLP,
\[
\begin{aligned}
& \min \sum_{i \in I} v_{i} \\
& \quad v_{i} \geq\left(2 a_{i} p_{i}+d_{i}\right) \bar{p}_{i}+\left(c_{i}-a_{i} \bar{p}_{i}^{2}\right) u_{i} \quad \bar{p} \in \cup_{i \in I} \mathcal{P}_{i}, \quad i \in I \\
& \quad(8.2 .2), \quad(8.2 .3), \quad(8.2 .4), \quad(8.2 .5) \\
& \quad v \in \mathbb{R}^{|I|}
\end{aligned}
\]

In this case we will create an ad-hoc structure to represent the semi-infinite perspective cuts (for the quadratic case) plus linear constraints.

Listing 8.15: Mixed-Integer Semi-Infinite Perspective Cuts (for the quadratic case) plus Linear Constraints
```

d_MISIPC_Q_LC_C :: d_LeafProblem_C
[
// u is treated as binary variable
// colsR = fact*colsl
dim_var -> [consL, colsR, colsl],
args -> [
// linear constraints input
ALi = d_vector(d_constant, [consL, colsl]),
ALr = d_vector(d_constant, [consL, colsR]),
rhsL = d_vector(d_constant, [consL]),
relsL= d_vector(d_rel, [consL]),
// cuts input
a = d_vector(d_constant, [colsR]),
d = d_vector(d_constant, [colsR]),
c = d_vector(d_constant, [colsl]),
// convex compacts input
Acc = d_vector(d_constant, [colsl, colsR]),
bcc = d_vector(d_constant, [colsl]),
// column relation factor
fact = d_constant,
// variables
x = d_vector(d_constant, [colsR]),
u = d_vector(d_constant, [colsl])
]
].

```

For this structure we could apply nice approximation techniques whereby we keep a finite perspective cut base, which is updated in an iterative fashion. This procedure is more efficient that dealing with the MINLP directly [71, 75].

Then based on the previous structure we can describe the following \(\operatorname{ARR}^{\Sigma}\), to reformulate a d_MIQP_SC_SEP_C into a d_MISIPC_Q_LC_C.

Listing 8.16: MIQP-SC-SEP to MISIPC_Q_LC
```

d_MIQP_SC_SEP_to_MISIPC_Q_LC_ARR : d_ARR_AIgebraic
[
A -> d_Composition_C(d_MILP_SC_C, d_Quadratic_F_C),
B -> d_MISIPC_Q_LC_C,
indexA -> [colsl=ii, colsR=ir],
indexB -> []
dimRel }->\mathrm{ - [consL = consL, colsR = colsR, colsl=colsl],
arg_map -> [
B..ALi = A..p1..Ai
B..ALr = A..p1..Ar,
B..rhsL = A...p1..rhs
B···relsL = A..p1..rels,
.a = A..p2···q,
B..d = \$((A..p1..cr(ir)+A..p2.c(ir))*A..p1..dir, [ir]),
B..c = \$(A..p1..ci(ii)*A..p1..dir, [ii]),
B..Acc = A..p1...Asc,
B···bcc = A..p1...bsc,
B..fact = A..fact
ans_map -> [
A.. x = B..x,
A..y = B···u
]
].

```

The perspective cuts and the MISOCP reformulations have proven to give better computational results than dealing with the MINLP directly [72].

\section*{Reformulation Diagram}

The previous structures and reformulations can be summed up by the diagram in Figure 8.2.


Figure 8.2: Convex Structures and Reformulations

\subsection*{8.3 Quadratic Variants}

In this section we will focus on variations of previously presented structure classes. These variations will consist on considering the objective functions to be quadratic. For instance we can make quadratic versions of d_KnapSack_C, d_MCF_C and d_MMCF_FC_C, which will mainly introduce the quadratic coefficient \(q\).

Listing 8.17: KnapSack Problem
```

d_QKnapSack_C :: d_LeafProblem_C
[
dim_var -> [E],
args -> [
x = d_vector(d_var, [E]),
= d_vector(d_constant, [E]), // quadratic cost
= d_vector(d_constant, [E]), // cost
m = d_constant // capacity
]
].

```

For the quadratic case of the continuous Knapsack problem (even if more complex than the non-quadratic one) there are several efficient methods for solving it [42, 138].

Listing 8.18: Quadratic Minimum Cost Flow
```

d_QMCF_C :: d_LeafProblem_C
[
dim_var }->\mathrm{ [N,E],
args -> [
SN = d_vector(d_constant, [E]), // start nodes
EN = d_vector(d_constant, [E]), // end nodes
SD = d_vector(d_constant, [N]), // supply/demand
cost = d_vector(d_constant, [E]), // cost per arc
q = d_vector(d_constant, [E]), // quadratic cost per arc
u = d_vector(d_constant, [E]), // arc capacity
flow = d_vector(d_var, [E]) // flow variables
],
dim_bound }->\mathrm{ [(SN,N), (EN,N)]
].

```

To solve a QMCF we can apply a CPLEX variant to solve quadratic network flow problems. We could also use a \(\epsilon\)-Relaxation algorithm [58] (or a parallel version [35]) that exploits the convexity and separability of QMCF. Moreover, we could apply PPRN [50], that was originally created to deal with non-linear Multicommodity MCFs.

We may also create a structure to represent a Quadratic MMCF(FC). However, the majority of the solution methods that could be applied to solve such a structure class are based on relaxations or other reformulations. Therefore, we will not create a structure class to explicitly represent a QMMCF(FC), on the other hand, we will use the composition structure d_Composition_C (cf. §7.1.1), with substructures d_MMCF_FC_C and d_Quadratic_F_C.

Note that this narrowing, d_Composition_C(d_MMCF_FC_C, d_Quadratic_F_C), since it has a specific semantical value (it represents a QMMCF(FC)), it allows us to register specific solvers that can deal with this particular decomposition. But more important, is the possibility of using this representation to create reformulation rules that conduct us to other, less intuitive, possible relaxations.

\subsection*{8.3.1 Reformulations}

Some of the previous structures may be only solvable by applying convenient relaxations or decompositions, which will be introduced in this section.

\section*{QMCF to MCF}

The difference between the QMCF and MCF structure classes is the inclusion of the quadratic terms in the objective function (OF). There is a way of obtaining an equivalent approximation of QMCF by linearizing the OF using a piece-wise linear function, and thus generating a MCF.

Note that QMCF's OF is separable in \(|\mathrm{E}|\) (arcs) terms of the form:
\[
f_{i j}\left(x_{i j}\right)=q_{i j} x_{i j}^{2}+c_{i j} x_{i j} \quad(i, j) \in \mathrm{E}
\]
where \(x_{i j}\) is bounded in the following way:
\[
0 \leq x_{i j} \leq u_{i j} \quad(i, j) \in \mathrm{E}
\]

Assume we select \(H_{i j}\) points for each arc, such that \(H_{i j} \subset\left[0 . . u_{i j}\right]\) then we can define a piece-wise linear function \(g_{i j}\) for each arc, such that,
\[
g_{i j}= \begin{cases}g_{i j}^{1}=f_{i j}\left(d_{i j}^{1}\right)+f_{i j}^{\prime}\left(d_{i j}^{1}\right)\left(x_{i j}-d_{i j}^{1}\right) & x_{i j} \in\left[0 . . \mathcal{I}\left(g_{i j}^{1}, g_{i j}^{2}\right)\right] \\ g_{i j}^{2}=f_{i j}\left(d_{i j}^{2}\right)+f_{i j}^{\prime}\left(d_{i j}^{2}\right)\left(x_{i j}-d_{i j}^{2}\right) & x_{i j} \in\left[\mathcal{I}\left(g_{i j}^{1}, g_{i j}^{2}\right) . . \mathcal{I}\left(g_{i j}^{2}, g_{i j}^{3}\right)\right] \\ \cdots & \\ g_{i j}^{\left|H_{i j}\right|}=f_{i j}\left(d_{i j}^{\left|H_{i j}\right|}\right)+f_{i j}^{\prime}\left(d_{i j}^{\left|H_{i j}\right|}\right)\left(x_{i j}-d_{i j}^{\left|H_{i j}\right|}\right) & x_{i j} \in\left[\mathcal{I}\left(g_{i j}^{\left|H_{i j}\right|-1}, g_{i j}^{\left|H_{i j}\right|}\right) . . u_{i j}\right]\end{cases}
\]
where \(d_{i j}^{h} \in H_{i j}, h \in\left[1 . .\left|H_{i j}\right|\right]\), and \(\mathcal{I}(\dot{g}, \ddot{g})\) is the x -axis of the intersection point between the linear functions \(\dot{g}\) and \(\ddot{g}\). Note that \(g_{i j}\) if convex, so we can generate an equivalent LP formulation without using auxiliary binary variables.

Assuming each \(g_{i j}^{h}=\hat{c}_{i j}^{h} x_{i j}+b_{i j}^{h}\), and that \(a_{i j}^{h}=\mathcal{I}\left(g_{i j}^{h}, g_{i j}^{h+1}\right)-\mathcal{I}\left(g_{i j}^{h-1}, g_{i j}^{h}\right)\), then the piece-wise linear function \(g_{i j}\) can be reformulated into the following equivalent LP,
\[
\begin{aligned}
n f_{i j}= & \sum_{h \in\left[1 . .\left|H_{i j}\right|\right]} \hat{c}_{i j}^{h} z_{i j}^{h} \\
& 0 \leq z_{i j}^{h} \leq a_{i j}^{h}
\end{aligned}
\]

If we see the \(z_{i j}^{h}\) as \(\left|H_{i j}\right|\) copies of the arc ( \(\mathrm{i}, \mathrm{j}\) ), having costs \(\hat{c}_{i j}^{h}\) and capacities \(a_{i j}^{h}\), and properly adding the flow conservation constraints for the variables \(z_{i j}^{h}\), then the minimization of \(\sum_{(i, j) \in|\mathrm{E}|} n f_{i j}\) subject to the constraints previously mentioned generates a MCF.

The solution of this MCF, stored in \(z_{i j}^{h}\), can be easily transformed back as an approximate solution of QMCF, by doing the following,
\[
x_{i j}=\sum_{h \in\left[1 . .\left|H_{i j}\right|\right.} z_{i j}^{h}
\]

Therefore we can obtain an arbitrary approximation of QMCF in terms of MCF by creating an arbitrary high number of parallel \(\operatorname{arcs}\left(H_{i j}\right)\).

This reformulation can not be expressed in terms of an ARR \({ }^{\Sigma}\). Thus, we need to create an \(A R R^{\mathcal{A}}\) that requires the implementation of the proper delegation solver that transforms the input of a QMCF into the input of a MCF, and the results of the MCF into the results of QMCF. This \(A R R^{\mathcal{A}}\) can be denoted by,

Listing 8.19: QMCF to MCF through linearization
```

d_QMCF_to_MCF_ARR : d_ARR_AIgorithmic
[
A }->\mathrm{ d d_QMCF_C,
B -> d_MCF_C
].

```

\section*{QMMCF(FC) Lagrangian Relaxations}

Using a mechanism analogous to the one applied for MMCF(FC) we can make a Flow and Knapsack relaxations of the QMMCF(FC) problem. Just that, instead of having MCF (or Knapsack) subproblems, these relaxations will have QMCF (or QKnapsack) subproblems. The detailed representation of these \(\mathrm{ARR}^{\Sigma}\) are herewith exposed. Note that to represent the \(\mathrm{QMMCF}(\mathrm{FC})\) problem we are using the narrowing d_Composition_C(d_MMCF_FC_C, d_Quadratic_OF_C).

Listing 8.20: QMMCF FC to Lagrangian Flow Relaxation
```

d_QMMCF_FC_to_Flow_Lagrangian_Relax_ARR : d_ARR_Algebraic
[
A -> d_Composition_C(d_MMCF_FC_C, d_Quadratic_F_C),
B }->\mathrm{ d_Lagrangian_Relax_C((d_QMCF_C, d_Simple_Selection_C), ? _ ),
indexA M [p1..E=(e,e1),p1···K=(k,k1),p1···N=n],
indexB -> []
dimRel -> [sub(d_QMCF_C)..N=p1..N, sub(d_QMCF_C)..E=p1..E,
sub(d_Simple_Selection_C )..d1=p1..E, linking..d1=p1..K*p1..E+p1..E,
Iinking..d2=2*p1.EE+p1..E*p1..K],
arg_map -> [
//_ MCF part
// Note that k is left untouched. Therefore k will be
// the free-index on which the QMCF structure will be replicated.
B..sub(d_QMCF_C)..SN = A..p1..SN,
B..sub(d_QMCF_C)..EN = A..p1..EN,
B..sub(d_QMCF_C)..q = A..p2..q
B···sub(d_QMCF_C) ..SD = \$(A···p1···SD(k,n), [n]),
B···sub(d_QMCF_C) ..cost = \$(A···p1···c(k,e)+A..p2···c((k,e)), [e]),

```
```

    B..sub(d_QMCF_C)..u = $(A..p1..b(k,e), [e]),
    //- Simple Selection part
    B...sub(d_Simple_Selection_C)..f = A..f,
    //-_Linear Constraints part (linking constraints)
    B..linking..A = $([ //left hand matrix
        // mutual arc capacity constraints \gamma
        $(-cs([1-> (e=e1), 0]), [e1, (e,k)]),
        $(0, [e1, e] )
    ],
        mutual arc capacity redundant constraints \alpha }\mp@subsup{\alpha}{i}{
        $(-cs([1-> (e=e1), 0]), [e1, (e,k)]),
        $(_cs([A\ldotsp1\ldotsu(d1)-> (e=e1), 0]), [e1, e] )
            ],
            [ // single capacity constraints }\mp@subsup{\beta}{i}{k
        $(-cs([1-> (e=e1,k=k1), 0]), [(e1,k1), (e,k)]),
        $(-cs([A\ldotsp1\ldotsb(k,d) -> (e=e1)]), [(e1,k), e] )
            ]]
            ]),
    B..linking.. rels = $([ // relations' vector
                $('=<','[e] ),
                $('=<', [(e,k)])
    B linking b ])
                ]),
                        =$([ // right hand side vector
            $(A..p1\ldotsu(a), [e] ),
            $(0, [e] )
            ])
    ans_map -> [
A..p1..flow = B..sub(d_QMCF_C)..flow
A..p1..desg = B..sub(d_Simple_Selection_C)..f
fixTI > [,
X = (B..sub(d_QMCF_C)..flow, B...sub(d_Simple_Selection_C )..f)
]

```
].

Listing 8.21: QMMCF FC to Lagrangian Knapsack Relaxation
```

d_QMMCF_FC_to_Knapsack_Lagrangian_Relax_ARR : d_ARR_AIgebraic
A -> d_Composition_C(d_MMCF_FC_C, d_Quadratic_F_C),
B -> d_Lagrangian_Relax_C(d_B_Ind_Replication_Selection_C(d_QKnapSack_C, ?_), ?_),
indexA -> [p1..E=e,p1.. K=(k,k1),p1..N=n],
indexB }->\mathrm{ [],
dimRel -> [b1..sub..E=p1..K, b1..sel..d1=p1..E,
linking..d1=p1..K*p1..E, linking..d2=p1..N*p1..K],
arg_map -> [
// Knapsack part
// Note that e is left untouched. Therefore e will be
// the free-index on which the QKnapsack structure will be replicated
B..sub..sub..c = \$(A..p1..c(k,e)+A..p2..c((k,e)), [k]),
B..sub..sub..w = \$(1, [k]),
B..sub..sub..q = \$(A...p2...q((k,e)), [k]),
B..sub..sub..m = A..p1..u,
//-_ Simple Selection part
B..sub..sel..f = A..p1..f,
//-_Linear Constraints part (linking constraints)
B..linking..A = \$([1-> (A..p1...SN(e)=n, k=k1),
-1->(A..p1..EN(e)=n, k=k1),
0]
[(n,k1), (e,k)]),
B.. linking...rels = \$('=', [(n,k)])
B..linking ..b = \$(A..p1..SD(k,n), [(k,n)]),
ans_map -> [
A..p1..flow = B..sub..sub..x,
A..p1..desg = B..sub ..sel..y'
fixTI -> [,
X = (B..sub..sub... ( B ..sub ..sub..y),
sub.. X = B..sub..sub... }x\mathrm{ ,
sub..Y = B..sub..sel..y
]
].

```

Note that these relaxations rely on the solution of well-known structures like d_QMCF_C and d_QKnapSack_C. These substructures can be transitively reformulated, by applying the proper ARRs. Again, this decomposition technique allows us to explore the space of compositions of solution methods in an automatic way (once the pieces are in place, of course).

\section*{QMMCF(FC) to MIQP-SC-SEP}

Transforming a QMMCF(FC) structure into MIQP-SC-SEP comes completely free. Since QMMCF(FC) is represented by the narrowing d_Composition_C(d_MMCF_FC_C, d_Quadratic_F_C) we could reformulate d_MMCF_FC_C into d_MILP_SC_C (cf. Listing 8.7), obtaining the composition
d_Composition_C(d_MILP_SC_C, d_Quadratic_F_C), which represents a MIQP-SC-SEP (cf. §8.2). Note that from MIQP-SC-SEP we can apply the \(\mathrm{ARR}^{{ }^{\text {S }}}\) d_MIQP_SC_SEP_to_MISIPC_Q_LC_ARR and/or d_MIQP_SC_SEP_to_MISOCP_ARR.

\subsection*{8.3.2 Final diagram}

By composing diagrams 8.1 and 8.2 and by adding these last reformulations we could build the diagram showed in Figure 8.3.


Figure 8.3: Full structure and reformulation diagram

\section*{Conclusions}

This thesis defines the grounds of a system which enables us to harness the vast body of knowledge that has been developed over the years about which combinations of (re)formulations and algorithms are best for many classes of optimization problems, and make it available to non experts. This involved the conception of a software system for automatically performing this task on behalf of, and transparently to, the user. This required defining an appropriate general concept of structured formulation (cf. Chapter 2) that be algorithmically treatable with appropriate tools, developing a large library of pre-defined structures which makes it easy for users to model their problems, link each of them with the appropriate solvers, and then be able to effectively search the huge space of possible formulations and solvers (with their many possible algorithmic parameters).

We described the set of architectural choices in the I-DARE system that have been designed to make an effective search possible, while avoiding to tie-in the system to specific search strategies that may not ultimately prove effective enough (such as complete enumeration). In particular, we discuss the fundamental role of the General Machine Learning Sub-system (GMLS), which allows to integrate general-purpose ML approaches with specialized methods for the (vastly) nontrivial task of computing the "objective function(s)" of the search. This task is "naturally" extended to that of selecting the best algorithmic configuration of the available solvers, thereby providing (whatever actual implementation of) the I-DARE(control) sub-system with a powerful tool to streamline the search. This requires a sophisticated ML (meta) process that is continuously running and keeps modifying the assessment of each reformulation with respect to given algorithms, so that it is kept synchronized with latest performance data given by practical problem solution runs. Although the use of ML techniques to select algorithmic parameters have very recently been advocated elsewhere, the scale of our proposal is, to the best of our knowledge, unheard of.

The outcome of this sophisticated process may well be a very significant improvement of the efficiency experienced by the "average" (non expert) user in the solution of her models, thereby significantly contributing to the overall scientific and technological progress. Furthermore, it has the possibility to substantially broadening the audience of the very many specialized solvers (and of their underlying theory) that have been developed over the last forty years for problems with specific structure. Indeed, insofar as such a system would greatly facilitate the fair comparison of solution algorithms and effective dissemination of the corresponding results, it might conceivably contribute to organizing, rationalizing and ultimately stimulating the research in solution algorithms for many classes of mathematical models. Actually, the possibility to taking into account monetary concerns during the search could lead to a substantial change in how mathematical software packages are evaluated, possibly forming the basis of a fair and extremely competitive "marketplace" for people supplying problems to be solved and people supplying solution algorithms to be used. Such a marketplace could dramatically improve the adoption of best-of-class approaches, possibly rewarding their authors in different ways, and it would allow the developers of very specialized approaches for very specific forms of structure to reach an audience that they would never be able to serve in the current system. This may radically change, for the better, the marketplace for mathematical software, while providing customers with much greater value. Therefore, while very significant theoretical and practical challenges still need to be overcome before this vision can become reality, we believe that the research on automatic reformulation and algorithm-selection techniques is worth to be undertaken.

\section*{C. 1 Perspectives on Deployment}

The I-DARE system aims at fostering a veritable breakthrough in the development, deployment and solution effectiveness of large-scale, difficult mathematical models. The catalyst of this paradigm shift is a software system capable of harnessing the vast body of knowledge that has been developed over the years about which combinations of (re)formulations, algorithms and algorithmic parameters (comprised the computing architectures) are best for each one in an almost endless list of applications, and make it available to non experts. This fundamentally hinges upon the concerted use of two main ingredients:
- machine learning approaches that continuously collects performance data from the solution of each problem, allowing the system to gradually build a database of unprecedented scale, granularity and informative content and making available to every interested party this very valuable data, currently dispersed in the scientific literature and unavailable outside a small community of experts;
- active involvement of academic researchers and of the industrial community dedicated to the development of solution algorithms, who have to equip the system with the best available numerical solvers and reformulation rules, together with all available expertise about the expected efficiency and effectiveness of a given solver on specific classes of problems.
Some vestigial aspects of the system in its foreseen final state can already be found in the NEOS server [128], which has been operating since 1999 and has served almost two millions of requests in its 10 years of life (with a growing trend: from about 17000 in 1999 to more than 225000 in 2009). Of course, a system capable of withstanding billions of requests and effectively learn from the huge amount of collected data would require a hugely different distributed architecture to start with, not to mention the many conceptual and technical advancements planned in this project. However, in view of I-DARE the NEOS server proves not so much as the technical feasibility of a system using remote computation for solving optimization problems, but especially the attractiveness for the users, especially the less optimization-savvy ones, of a system which does not require local installation of complex solution systems (possibly with many cross-dependencies, licensing issues, and more), in spite of the fact that it is still completely up to the user to write the model in the appropriate format and select the appropriate solver among the feasible ones. This underlines how the ultimate success of the system crucially depends on appropriate use of social intelligence techniques to construct a positive feedback loop whereby all involved parties have an incentive to use the system, which in turn makes the system better. In particular:
- End-users will have to find an easy-to-use system which provides them structured (e.g. objectoriented) means for rapidly and correctly write down mathematical models. The system should present them with a vast library of ready-to-use structures to choose from, thereby sparing them with the need to re-invent the wheel. The structures should be presented in quite abstract terms (i.e. by specifying their semantic meaning, rather than how this is actually obtained), thereby isolating them from a part of the complexity of a modeling exercise. The system should also consistently provide them with better-or at least comparablesolution results w.r.t. those obtainable by a completely hand-made solution, in a fraction of the time required for the latter and without requiring specific modeling or algorithmic expertise. The system will have to be open and accessible, with very low entry cost barriers (it should be basically free for academic purposes) and capable of accommodating different usage patterns for the needs of different applications domains. As a consequence, end-users will have considerable incentives to use the system, thereby providing its machine learning components with a growing set of data, that in turn will make the system more and more efficient and therefore attractive.
- Academic researchers in the field of mathematical models and algorithms development will have to find in the system the perfect benchmarking machine, capable of providing them with unprecedented amounts of data about the performances of algorithms on classes of models.

Currently, experimentally proving the efficiency and effectiveness of a new or improved algorithm requires a considerable effort not only for devising and implementing them, but also for collecting a set of significant testing instances, collecting all (or at least a reasonable set of) possible competing algorithms, set up a level-playing field which allows to fairly compare their performances (factoring out issues like the appropriate setting of the computational parameters), and finally perform the comparison, which can be in itself a long and highly computationally demanding task. At the cost of subjecting themselves to the required interface of the system (which, therefore, will have to be flexible and easy to set up) the researchers will be spared a very substantial amount of the necessary work to evaluate the efficiency and effectiveness of their creations, in addition to having accurate data about the behavior of the current solution methods, which may indicate their shortcomings and therefore suggests the most promising avenues for improvements. The nature of the system will guarantee the community that all comparisons be performed fairly, thereby substantially increasing the level of confidence in them, and motivating the top researchers to continuously provide the system with improved versions of their algorithms to earn the corresponding recognition. This will provide end-users with a constantly improving set of numerical solvers, an improvement they contributed to by sharing problem and solution data. Academia will also have to be at the forefront of the meta-learning problem-that is, providing the most appropriate machine learning approaches for choosing the best (formulation, algorithm, configuration) triplet, which is in itself a very complex task that will likely require the combination of several different techniques for classification and regression, data mining, artificial intelligence, and search in very large structured spaces. Finally, the many researchers currently working on finding the most appropriate formulations for different applications will be strongly motivated to provide the reformulation rules that will be used to search the formulations space, as the existence of a system that will bring to near-immediate fruition of their work to the large number of potentially interested end-users will substantially increase the relevance of their research subject.
- Commercial providers of numerical software for mathematical models could somewhat oppose the process, insomuch as it pushes towards a fair and extremely competitive marketplace where the cost/efficiency ratio of each software package, rather than other marketing factors, is basically the only considered performance measure. For this to happen, the system will have to provide the option to select the solution approach taking into account economic factors such as total cost of solution, comprising both CPU time cost and software licensing costs. However, providers of best-of-class approaches will have the opportunity to show off top performances, and therefore retain high shares of usage. Furthermore, as the system substantially improves the adoption of quantitative approaches far beyond the current set of users, any loss in market share can be more than made up with the increase in the users base; said otherwise, as the system gains traction not fully supporting it would imply losing out a large and growing set of potential users. Therefore, it can be expected that commercial providers will find it convenient, and ultimately necessary, to provide the system with the best of their worth. Indeed, it should be expected that the system actually substantially increase the set of available numerical solvers. This is because, currently, only a few general-purpose solvers have a large enough audience to be commercially viable. This is especially true since, apart from the theoretical and implementation issues inherent to devising a state-of-the-art commercial solver, a perspective entrepreneur also has to consider market positioning and the expenses necessary to educate users in order to breaking into a niche and fragmented market. With a system in place which is capable of selecting specialized algorithms on behalf of the users, a very small per-use licensing fee rewarded to algorithms creators may rapidly make up very substantial sums, especially if the user base rapidly swells, thereby providing ample incentives to developers to chase important structures. In particular, the fact that the system will be able to combine different solution algorithms in unprecedented ways, using such methods as decomposition approaches, will make it possible for very efficient, specialized codes for models with very specific structures to be used within approaches for
the very many, much larger-scale problems containing these as substructures (in tasks such as bounds computation, heuristics search, valid inequalities separation and many others). Thus, a much more vibrant ecosystem for specialized optimization approaches will be born and thrive, whereby commercial providers will use the system's data to gauge the economic opportunity to invest in the development of numerical solvers for specific structures, and market forces will drive competition to provide ever-increasing efficiency to end-users, abating market inefficiencies and costs due to marketing, user education, user retention and other non-technical issues of this kind. Academic providers of open-source, free numerical solvers, motivated by recognition rather than by economic incentives, will also constitute a powerful force to ensure continuous improvements in the performances, as well as zero-cost less-effective approaches for those many applications where top-performing solutions are not necessary.

\section*{C. 2 I-DARE challenges}

As previously illustrated, the I-DARE system ultimately aims at changing the way in which mathematical models of reality are developed and deployed. While at the beginning the system will target a narrow set of structures, mostly coming from decision and optimization problems arising in scientific and industrial applications, the system is conceptually open to integration of very different sets of mathematical components from almost all fields of human speculative and practical activities.

One of the main challenges currently facing scientific and technological development is that the explosive growth of information is rapidly overcoming humans' capabilities to comprehend and synthesize the available knowledge, which is the fundamental step for being able to produce new ideas. The current forms of reaction are a growing specialization, with each research field continuously producing initially narrow sub-fields which later on grow and possibly further subdivide, and teaming of specialists of different disciplines to tackle the same problem. Both approaches, although workable, have clear limitations: specialization leads to lack of wider cultural view, with possibly related research being carried on independently in several different fields because of lack of communication, and human interactions do not scale well after a handful of people, unless supported by specific organizations (which however have their own cost).

ICT systems have clearly played a major role in allowing the pace of scientific and technological process to continue unabated: collaboration tools like e-mail, digital libraries with the available sophisticated search tools, and open distribution of software have all provided enormous benefits to the efficiency of the research and development processes. However, these tools ultimately rely on human knowledge and intervention to make connections and fill-in the gaps. Just as information search eventually had to move away from human scanning of unsorted data and/or human-based categorization to fully embrace algorithmic approaches, the only ones capable of coping with growth of available information, it can be foreseen that more sophisticated ICT-based approaches will eventually be needed to cope with the explosive growth of available scientific theories and corresponding mathematical tools in a way that may simply not be possible to humans. Clearly, fundamental innovation will always - or at least for the foreseeable future - be a product of human's intuition and imagination. However, like a web search stitches together a relevant set of human-produced pieces of information in a way that the original creators of that information may have never imagined, selecting and gluing together the appropriate set of individual components to tackle one specific problem in a previously unforeseen way, may well become predominantly a work for appropriate algorithms.

The analogy with search should not be taken too far: the tasks of the I-DARE system are on one hand far more complex, requiring the exploration of a huge space of configurations and the estimation of the expected performances of each, and on the other hand far less complex as the underlying universe is far smaller and immensely more structured. However, just like search has revolutionized the use of Internet and made it almost overnight an incredibly useful tool outside the small niche of specialists which were previously able to deal with it, automatic search methods like I-DARE have the potential to revolutionize the use of mathematical models, making it possible
and convenient for a much larger audience than nowadays conceivable. The potential impact of this on the productivity of our society could really be huge.

Another absolutely nontrivial effect of a system like I-DARE is the fundamental push towards standardization and immediate re-usability of the results. In the academic world, purportedly the epitome of openness and availability of information, there is often too little incentive to make one's creations available to the community in a way that allows for immediate re-use from other interested parties. While development of solution codes, models or techniques does bring recognition, making it easy for others to access and use them requires a further substantial work which is often not valued enough, if at all, at the various levels where the impact of a research is judged. From the commercial viewpoint, all ICT history can be read as the continuous struggle between standardbased and proprietary solutions, which reveals how the huge technical superiority of a system where components can be sourced from different vendors and easily swapped-in is not always sufficient to battle the enormous incentive to tie-in customers to one's organization. Disclosing just about the minimum necessary amount of information and/or artificially segmenting the market along purposely created incompatibility lines are always strong temptations, all the more in a world where rampant competition makes it very difficult for anybody to carve his niche. It can be argued that the current system may not contain enough forces to steer evolution of the research and industrial behavior towards the most open possible collaboration; rather, a more successful researcher or entrepreneur may be he who subtly plants enough barriers to keep others to fully appreciate and make sense of his developments, artificially conserving one's competitive advantage. While this may be beneficial to the individual, it is generally detrimental for the society. Requiring the results-be it models, reformulation methods or solution software - to be available for immediate use as a prerequisite for them earning credits in the first place, a system like I-DARE may help in providing some of the missing incentives, harnessing the power of market forces in the conditions where it gives its best, i.e., in a fair and fiercely competitive market.

Perhaps the most important possible outcome of the project may not even be the I-DARE system itself, but the proof that systems streamlining and automatizing exploitation of scientific and technological advances through the use of sophisticated ICT tools is possible. While I-DARE is designed for being extremely open and capable to evolve, it may later be proven that some of its fundamental design decisions and concepts can be subsumed by even more powerful and general mechanisms. Unleashing a flurry of research about tools that make research and collaboration more productive may actually be the most relevant long-lasting effect that historians of science might credit the I-DARE project for in a far future. It is even unclear at this stage whether it is the technological aspect (finding the right algorithms and the appropriate system structure) that provides the most relevant contribution, or rather it is the social intelligence aspects of the project that ultimately proves to be the fundamental force which drives change; for science and technology provide most useful to society when they find ways to better satisfy the fundamental human needs to explore, share, and communicate. Thus, aspects like social acceptance among the interested communities, dissemination, and even politics and regulatory matters at the highest level may one day become fundamental "make or break" points for the possible future development of the idea.

\section*{Bibliography}
[1] COIN-OR, Computational infrastructure for operations research. http://www.coin-or.org.
[2] CSDP A Library for Semidefinite Programming. http://infohost.nmt.edu/~borchers/ csdp.html.
[3] DIMACS minimum cost flow problems. http://lpsolve.sourceforge.net/5.5/DIMACS_ mcf.htm.
[4] FLOPC ++ (Formulation of Linear Optimization Problems in \(C++\) ). https://projects.coinor.org/FlopC++.
[5] GAMS Solver Descriptions. http://www.gams.com/solvers/solvers.htm.
[6] GLOBAL Library. http://www.gamsworld.org/global/globallib.htm.
[7] IBM ILOG CPLEX. http://www-01.ibm.com/software/integration/optimization/ cplex/.
[8] LOQO Users Manual. http://www.princeton.edu/~rvdb/tex/loqo/loqo405.pdf.
[9] The MOSEK Optimization Software. http://www.mosek.com/.
[10] Ontoprise. http://www.ontoprise.de/.
[11] SDPA. http://sdpa.indsys.chuo-u.ac.jp/sdpa/.
[12] Stochastic Modeling Interface. http://www.coin-or.org/projects/Smi.xml.
[13] The Canad problems. http://www.di.unipi.it/di/groups/optimize/Data/MMCF.html\# Canad.
[14] LEMON: Library for Efficient Modeling and Optimization in Networks. http://lemon.cs.elte.hu/trac/lemon/, May 2010.
[15] D. Kendrick A. Brook and A. Meeraus. Gams, a user's guide, 1988.
[16] Tobias Achterberg. Scip - a framework to integrate constraint and mixed integer programming. Technical report, Konrad-Zuse-Zentrum fr Informationstechnik Berlin, 2004.
[17] Tobias Achterberg, Thorsten Koch, and Alexander Martin. MIPLIB 2003. Operations Research Letters, 34(4):361-372, 2006.
[18] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Network Flows: Theory, Algorithms and Applications. Prentice Hall, 1993.
[19] Ahuja, R.K., Magnanti, T.L., and Orlin, J.B. Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Englewood Cliffs, 1993.
[20] S. Aktürk, A. Atamtürk, and S. Gürel. A strong conic quadratic reformulation for machinejob assignment with controllable processing times. Operations Research Letters, 37(3):187191, 2009.
[21] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. The Traveling Salesman: a Computational Study. Princeton University Press, Princeton, 2007.
[22] K. R. Apt. Principles of Constraint Programming. Cambridge University Press, 2003.
[23] C. Audet, J. Brimberg, P. Hansen, S. Le Digabel, and N. Mladenović. Pooling problem: Alternate formulations and solution methods. Princeton University Press, Princeton, 50(6):761776, 2004.
[24] C. Audet, P. Hansen, B. Jaumard, and G. Savard. Links between linear bilevel and mixed 0-1 programming problems. Journal of Optimization Theory and Applications, 93(2):273-300, 1997.
[25] C. Audet, P. Hansen, F. Messine, and S. Perron. The minimum diameter octagon with unit-length sides: Vincze's wife's octagon is suboptimal. Journal of Combinatorial Theory A, 108(1):63-75, 2004.
[26] E. Balas and E. Zemel. Facets of the knapsack polytope from minimal covers. SIAM Journal on Applied Mathematics, 34(1):119-148, January 1978.
[27] Egon Balas, Sebastián Ceria, and Gérard Cornuéjols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. Math. Program., 58(3):295-324, 1993.
[28] M. Ball, T. Magnanti, C. Monma, and G. Nemhauser. Network Routing, volume 8 of Handbooks in Operations Research and Management Science. North-Holland, Amsterdam, 1995.
[29] C. Barnhart and G. Laporte. Transportation, volume 14 of Handbooks in Operations Research and Management Science. North-Holland, Amsterdam, 2007.
[30] H. Ben Amor, J. Desrosiers, and A. Frangioni. On the Choice of Explicit Stabilizing Terms in Column Generation. Discrete Applied Mathematics, 157(6):1167-1184, 2009.
[31] A. Ben-Tal and A. Nemirovski. Lectures on Modern Convex Optimization: Analysis, Algorithms, Engineering Applications. MPS-SIAM Series on Optimization. SIAM, Philadelphia, 2001.
[32] A. Ben-Tal and A. Nemirovski. Selected topics in robust convex optimization. Mathematical Programming, 112(1):125-158, 2008.
[33] J. F. Benders. Partitioning procedures for solving mixed-variables programing problems. Numerische Mathematik, 4:238-252, 1962.
[34] Frèdèric Benhamou. Interval constraint logic programming. In Andreas Podelski, editor, Constraint Programming: Basic Trends. Springer, 1994.
[35] P. Beraldi, F. Guerriero, and R. Musmanno. Parallel algorithms for solving the convex minimum cost flow problem. Comput. Optim. Appl., 18(2):175-190, 2001.
[36] D.P. Bertsekas and P. Tseng. Relax-iv: a faster version of the relax code for solving minimum cost flow problems. Technical report, Dept. of Electrical Engineering and Computer Science, MIT, 1994.
[37] G. Bigi, A. Frangioni, and Q.H. Zhang. Outer Approximation Algorithms for Canonical DC Problems. Journal of Global Optimization, to appear, 2009.
[38] C.M. Bishop. Pattern recognition and machine learning. Springer, New York, 2006.
[39] J. Bjorkqvist and T. Westerlund. Automated reformulation of disjunctive constraints in minlp optimization. Computers and Chemical Engineering, 23:S11-S14, 1999.
[40] Pierre Bonami, Lorenz T. Biegler, Andrew R. Conn, Grard Cornujols, Ignacio E. Grossmann, Carl D. Laird, Jon Lee, Andrea Lodi, Franois Margot, Nicolas Sawaya, and Andreas Wchter. An algorithmic framework for convex mixed integer nonlinear programs. Discrete Optimization, 5(2):186-204, 2008. In Memory of George B. Dantzig.
[41] S. Boyd and L. Vandenberghe. Convex Optimization. Cambridge University Press, 2004.
[42] Peter Brucker. An o(n) algorithm for quadratic knapsack problems. Operations Research Letters, 3(3):163-166, 1984.
[43] G. Calafiore and M. Campi. Uncertain convex programs: randomized solutions and confidence levels. Mathematical Programming, 102(1):25-46, 2005.
[44] Gilles Caporossi and Pierre Hansen. Variable neighborhood search for extremal graphs: 1. the autographix system. Discrete Mathematics, 212(1-2):29-44, 2000.
[45] P. Cappanera and A. Frangioni. Symmetric and Asymmetric Parallelization of a CostDecomposition Algorithm for Multi-Commodity Flow Problems. INFORMS Journal on Computing, 15(4):369-384, 2003.
[46] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. Management Science, (35):164-176, 1989.
[47] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. European Journal of Operational Research, (78):146-161, 1994.
[48] Y. Caseau and F. Laburthe. Improved clp scheduling with task intervals. In Proceedings of the Eleventh International Conference on Logic Programming (ICLP 1994), pages 369-383. The MIT Press, 1994.
[49] A. Cassioli, D. Di Lorenzo, M. Locatelli, F. Schoen, and M. Sciandrone. Machine Learning for Global Optimization. Technical Report 2360, Optimization Online, 2009.
[50] J. Castro and N. Nabona. An Implementation of Linear and Nonlinear Multicommodity Network Flows. EJOR, 92:37-53, 1996.
[51] Jordi Castro. A specialized interior-point algorithm for multicommodity network flows. SIAM Journal on Optimization, 10:852-877, 2000.
[52] Sebastián Ceria and Jo ao Soares. Convex programming for disjunctive convex optimization. Mathematical Programming, 86:595-614, 1999.
[53] Marco Colombo, Andreas Grothey, Jonathan Hogg, Kristian Woodsend, and Jacek Gondzio. A Structure-Conveying Modelling Language for Mathematical and Stochastic Programming. Technical report, School of Mathematics and Maxwell Institute, University of Edinburgh, Edinburgh, Scotland, March.
[54] ILOG Concert Technology. http://www.ilog.com/products/optimization/tech/ concert.cfm.
[55] Teodor Gabriel Crainic, Michel Gendreau, and Judith M. Farvolden. A simplex-based tabu search method for capacitated network design. INFORMS J. on Computing, 12(3):223-236, 2000.
[56] Claudia D'Ambrosio, Antonio Frangioni, Leo Liberti, and Andrea Lodi. Experiments with a feasibility pump approach for nonconvex minlps. In SEA10 Proceedings, LNCS, 2010.
[57] G. B Dantzig. Linear programming and extensions. Princeton, NJ: Princeton University Press, 1963.
[58] Renato De Leone, Robert R. Meyer, and Armand Zakarian. A partitioned \(\epsilon\) - relaxation algorithm for separable convex. Comput. Optim. Appl., 12(1-3):107-126, 1999.
[59] Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors. Column generation. Springer, 2005.
[60] M. Dorigo and C. Blum. Ant colony optimization theory: A survey. Theoretical Computer Science, 344(2-3):243-278, 2005.
[61] O. du Merle, P. Hansen, B. Jaumard, and N. Mladenović. An interior point algorithm for minimum sum-of-squares clustering. SIAM Journal Scientific Computing, 21(4):1485-1505, 2000.
[62] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM, 19(2):248-264, 1972.
[63] William B. Ewald. From Kant to Hilbert: A Source Book in the Foundations of Mathematics, 2 vols. Oxford Uni. Press, 1996.
[64] A. Lodi F. Focacci and M. Milano. Cost-based domain filtering. In J. Jaffar, editor, Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming, LNCS 1713, pages 189-203. Springer-Verlag, 1999.
[65] A. Lodi F. Focacci and M. Milano. Cutting planes in constraint programming an hybrid approach. In R. Dechter, editor, Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, LNCS 1894, pages 187-201. Springer-Verlag, 2000.
[66] R. Fourer and D. Gay. The AMPL Book. Duxbury Press, Pacific Grove, 2002.
[67] B. R. Fox. Chronological and non-chronological scheduling. In Proceedings of the First Annual Conference on Artificial Intelligence: Simulation and Planning in High Autonomy Systems. Tucson, Arizona, 1990.
[68] A. Frangioni. On a New Class of Bilevel Programming Problems and its Use For Reformulating Mixed Integer Problems. European Journal of Operations Research, 82:615-646, 1995.
[69] A. Frangioni and B. Gendron. 0-1 Reformulations of the Multicommodity Capacitated Network Design Problem. Discrete Applied Mathematics, 157(6):1229-1241, 2009.
[70] A. Frangioni and C. Gentile. New Preconditioners for KKT Systems of Network Flow Problems. SIAM Journal on Optimization, 14(3):894-913, 2004.
[71] A. Frangioni and C. Gentile. SDP Diagonalizations and Perspective Cuts for a Class of Nonseparable MIQP. Operations Research Letters, 35(2):181-185, 2007.
[72] A. Frangioni and C. Gentile. A computational comparison of reformulations of the perspective relaxation: Socp vs. cutting planes. Operations Research Letters, 37(3):206-210, 2009.
[73] A. Frangioni, M.G. Scutellà, and E. Necciari. A Multi-exchange Neighborhood for Minimum Makespan Machine Scheduling Problems. Journal of Combinatorial Optimization, 8:195-220, 2004.
[74] A. Frangioni and S. Serra Capizzano. Spectral Analysis of (Sequences of) Graph Matrices. SIAM Journal on Matrix Analysis and Applications, 23(2):339-348, 2001.
[75] Antonio Frangioni and Claudio Gentile. Perspective cuts for a class of convex 0-1 mixed integer programs. Math. Program., 106(2):225-236, 2006.
[76] G12: A Constraint Programming Platform. http://www.g12.csse.unimelb.edu.au/.
[77] A. M. Geoffrion. Generalized benders decomposition. Journal of Optimization Theory and Applications, 10:237-260, 1972.
[78] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. J. ACM, 36(4):873-886, 1989.
[79] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by successive approximation. Math. Oper. Res., 15(3):430-466, 1990.
[80] D.E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, Reading, MA., 1989.
[81] E. Gourdin, P. Hansen, and B. Jaumard. Finding maximum likelihood estimators for the three-parameter weibull distribution. Journal of Global Optimization, 5(4):373-397, 1994.
[82] M. Grant, S. Boyd, and Y. Ye. Disciplined convex programming, pages 155-210. Liberti and Maculan [115].
[83] J. Gondzio A. Grothey. Exploiting structure in parallel implementation of interior point methods for optimization. Technical report, School of Mathematics, University of Edinburgh, Scotland, December 2004.
[84] M. Guignard. Lagrangean Relaxation. TOP, 11(2):151-228, 2003.
[85] P. Hansen, J. Brimberg, N. Mladenović, and D. Urosević. Primal-dual variable neighbourhood search for the simple plant location problem. INFORMS Journal on Computing, 19(4):552564, 2007.
[86] P. Hansen and B. Jaumard. Cluster analysis and mathematical programming. INFORMS Journal on Computing, 79:191-215, 1997.
[87] P. Hansen and N. Mladenović. Variable neighbourhood search: Principles and applications. European Journal of Operations Research, 130:449-467, 2001.
[88] R. Hemmecke, M. Köppe, J. Lee, and R. Weismantel. Nonlinear integer programming. In M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, and L. Wolsey, editors, 50 Years of Integer Programming 1958-2008: The Early Years and State-of-the-Art Surveys, Studies in Computational Intelligence. Springer-Verlag, Berlin, 2009.
[89] M. Hinze, R. Pinnau, M. Ulbrich, and S. Ulbrich. Optimization with PDE Constraints. Springer, 2009.
[90] J. N. Hooker. A hybrid method for planning and scheduling. Constraints, (10):385-401, 2005.
[91] J. N. Hooker. Logic-based modeling. In L. Pitsoulis G. Appa and H. P. Williams, editors, Handbook on Modelling for Discrete optimization, pages 61-102. Springer, 2006.
[92] J. N. Hooker and G. Ottosson. Logic-based benders decomposition. Math. Program, (96):3360, 2003.
[93] John Hooker. Logic-Based Methods for Optimization, Combining Optimization and Constraint Satisfaction. John Wiley \& Sons, Inc, 2000.
[94] John Hooker. Integrated Methods for Optimization. 2007.
[95] R. Horst and N. V. Thoai. Dc programming: Overview. Journal of Optimization Theory and Applications, 103(1):1-43, October 1999.
[96] F. Hutter, H.H. Hoos, K. Leyton-Brown, and T. Stützle. Paramils: An automatic algorithm configuration framework. Journal of Artificial Intelligence Research, 36:267-306, 2009.
[97] IBM. Passing your model using mathematical programming system (MPS) format. http://www-306.ibm.com/software/data/bi/osl/pubs/Library/featur11.htm, 2003.
[98] A. Frangioni J. Castro. A parallel implementation of an interior-point algorithm for multicommodity network flows. In Vector and Parallel Processing - VECPAR 2000: 4th International Conference, Porto, Portugal, June 2000. Selected Papers and Invited Talks, volume 1981 of Lecture Notes in Computer Science, page 301. Springer Berlin / Heidelberg, 2001.
[99] B. Jaumard, P. Hansen, and M. Poggi de Aragão. Column generation methods for probabilistic logic, pages 313-331. IPCO, University of Waterloo Press, 1990.
[100] David S. Johnson. Approximation algorithms for combinatorial problems. In STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing, pages 38-49, New York, NY, USA, 1973. ACM.
[101] J. Judice and G. Mitra. Reformulation of mathematical programming problems as linear complementarity problems and investigation of their solution methods. Journal of Optimization Theory and Applications, 57(1):123-149, 1988.
[102] Michael Kifer, Georg Lausen, and James Wu. Logical fundations of object-oriented and frame-based languages. Technical report, Department of Computer Science, SUNY at Stony Brook, NY, 1994.
[103] Morton Klein. primal method for minimal cost flows with applications to the assignment and transportation problems. MANAGEMENT SCIENCE, 14(3):205-220, November 1967.
[104] M. Kočvara and J.V. Outrata. Optimization problems with equilibrium constraints and their numerical solution. Mathematical Programming, 101(1):119-149, 2004.
[105] S. Kucherenko, P. Belotti, L. Liberti, and N. Maculan. New formulations for the kissing number problem. Discrete Applied Mathematics, 155(14):1837-1841, 2007.
[106] Steve Homer L. Fortnow. A short history of computational complexity. In J. Dawson D. van Dalen and A. Kanamori, editors, The History of Mathematical Logic. North-Holland, Amsterdam, 2002/2003.
[107] M. Labbé, D. Peeters, and J.-F. Thisse. Location on networks. Network Routing, volume 8 of Handbooks in Operations Research and Management Science. North-Holland, Amsterdam, 1995.
[108] C. Lavor, L. Liberti, and N. Maculan. Molecular distance geometry problem. Encyclopedia of Optimization, Springer, New York, 2 edition, 2009.
[109] C. Lavor, L. Liberti, N. Maculan, and M.A. Chaer Nascimento. Solving Hartree-Fock systems with global optimization metohds. Europhysics Letters, 5(77):50006p1-50006p5, 2007.
[110] C. Kun Lee, A. Singer, and P. Barton. Global optimization of linear hybrid systems with explicit transitions. Systems \& Control Letters, 51(5):363-375, 2004.
[111] Claude Lemarchal. Lagrangian relaxation. Technical report, Inria, 655 avenue de l'Europe, Montbonnot, 38334 Saint Ismier, France, 2002.
[112] L. Liberti. Reformulation techniques in mathematical programming, in preparation. Thèse d'Habilitation à Diriger des Recherches, Université Paris IX.
[113] L. Liberti. Reformulations in mathematical programming: Definitions and systematics. RAIRO-RO, 43(1):55-86, 2009.
[114] L. Liberti, S. Cafieri, and F. Tarissan. Reformulations in mathematical programming: a computational approach. In A. Abraham, A.-E. Hassanien, P. Siarry, and A. Engelbrecht, editors, Foundations of Computational Intelligence Vol. 3, number 203 in Studies in Computational Intelligence, pages 153-234. Springer, Berlin, 2009.
[115] L. Liberti and N. Maculan. Global Optimization: from Theory to Implementation. Springer, Berlin, 2006.
[116] L. Liberti, N. Maculan, and Y. Zhang. Optimal configuration of gamma ray machine radiosurgery units: the sphere covering subproblem. Optimization Letters, 3:109-121, 2009.
[117] L. Liberti and C.C. Pantelides. An exact reformulation algorithm for large nonconvex nlps involving bilinear terms. Journal of Global Optimization, (36):161-189, 2006.
[118] D. S. Johnson M. R. Garey and L. Stockmeyer. Some simplified np-complete problems. In STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing, pages 47-63, New York, NY, USA, 1974. ACM.
[119] C. Maranas and C. Floudas. Global optimization in generalized geometric programming. Computers and Chemical Engineering, 21(4):351-369, 1997.
[120] F. Marinelli, O. de Weck, D. Krob, and L. Liberti. A general framework for combined moduleand scale- based product platform design. Technical report, LIX, Ecole Polytechnique, 2007.
[121] K. Marriott and P. J. Stuckey. Programming with constraints: an introduction. MIT Press, 1998.
[122] J.R.R.A. Martins, C. Marriage, and N. Tedford. pyMDO: An Object-Oriented Framework for Multidisciplinary Design Optimization. ACM Transactions on Mathematical Software, 36(4):Article No. 20, 2009.
[123] Ted Ralphs Matthew Saltzman, Lszlo Ladnyi. The COIN-OR Open Solver Interface: Technology Overview, May 2004.
[124] The MCFClass Project. http://www.di.unipi.it/optimize/Software.
[125] Multicommodity Problems. http://www.di.unipi.it/optimize/Data/MMCF.html.
[126] G. Moser and R. Zach. The epsilon calculus. In Computer Science Logic, volume 2803 of Lecture Notes in Computer Science, page 455. Springer Berlin / Heidelberg, 2003.
[127] George L. Nemhauser and Laurence A. Wolsey. Integer and Combinatorial Optimization. John Wiley \& Sons, New York, 1988.
[128] NEOS: Network-Enabled Optimization Server. http://neos.mcs.anl.gov/.
[129] Ulf Nilsson and Jan Maluszyński. Logic, Programming and PROLOG (2ED). John Wiley \& Sons Ltd., 1995.
[130] OptimJ. http://www.ateji.com/optimj.html, 2009.
[131] I. Papamichail and C. Adjiman. A rigorous global optimization algorithm for problems with ordinary differential equations. Journal of Global Optimization, 24:1-33, 2002.
[132] C. Le Pape. Implementation or resource constraints in ilog schedule: A library for the development of constraint-based scheduling systems. Intelligent Systems Engineering, (3):5566, 1994.
[133] Benjamin C. Pierce. Types and programming languages. MIT Press, Cambridge, MA, USA, 2002.
[134] puLP. http://code.google.com/p/pulp-or/.
[135] I. Quesada and I.E. Grossmann. An LP/NLP based branch and bound algorithm for convex MINLP optimization problems. Computers \& Chemical Engineering, 16(10-11):937 - 947, 1992. An International Journal of Computer Applications in Chemical Engineering.
[136] Rima: A Tool for Math Modelling. http://www.incremental.co.nz/projects/rima/, 2009.
[137] Kipp Martin Robert Fourer, Jun Ma and Wayne Sheng. Optimization Services 1.0 User's Manual, November 2007.
[138] A. G. Robinson, N. Jiang, and C. S. Lerme. On the continuous quadratic knapsack problem. Math. Program., 55(1-6):99-108, 1992.
[139] J. M. Robson. Algorithms for maximum independent sets. J. Algorithms 7, pages 425-440, 1986.
[140] SDPA File Format. http://euler.nmt.edu/~brian/sdplib/FORMAT.
[141] B. Settles. Active Learning Literature Survey. Computer Sciences Technical Report 1648, University of Wisconsin-Madison, 2009.
[142] .D. Sherali and W.P. Adams. A Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems. Kluwer Academic Publishers, Dodrecht, 1999.
[143] H. Sherali. Personal communication. 2007.
[144] E.M.B. Smith and C.C. Pantelides. A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex minlps. Computers and Chemical Engineering, 23:457-478, 1999.
[145] S. Sonnenburg, G. Raetsch, C. Schaefer, and B. Schoelkopf. Large Scale Multiple Kernel Learning. Journal of Machine Learning Research, (7):1531-1565, 2006.
[146] Robert A. Stubbs and Sanjay Mehrotra. A branch-and-cut method for 0-1 mixed convex programming. Mathematical Programming, 86(3):515-532, 1999.
[147] Terrance Swift and David S. Warren. The XSB System Version 3.2.
[148] R. E. Tarjan and A. E. Trojanowski. Finding a maximum independent set. Technical report, Stanford University, June 1976.
[149] Tawarmalani, Mohit, Sahinidis, and Nikolaos V. Semidefinite relaxations of fractional programs via novel convexification techniques. J. of Global Optimization, 20(2):133-154, 2001.
[150] E. Tomita and T. Seki. Discrete Mathematics and Theoretical Computer Science, volume 2731 of Lecture Notes in Computer Science, chapter An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique, pages 278-289. Springer Berlin / Heidelberg, 2003.
[151] A. Turing. On computable numbers, with an application to the entscheidungsproblem. In Reprinted in M. Davis (ed.), editor, London Mathematical Society, Series 2, Volume 42 (1939). The Undecidable, Raven Press, Hewlett, NY, 1965.
[152] E. Pesch U. Dorndorf and T. Phan-Huy. Solving the open shop scheduling problem. Journal of Scheduling, (4):157-174, 2001.
[153] Jean van Heijenoort. A Source Book in Mathematical Logic, 1879-1931. Harvard Univ. Press, 1967.
[154] T.J. van Roy and L.A. Wolsey. Solving mixed integer programming problems using automatic reformulation. Operations Research, 35(1):45-57, 1987.
[155] J. von Neumann. The principles of large-scale computing machines. IEEE Annals of the History of Computing, 10(4):243-256, October-December 1988.
[156] Arthur Westerberg. Ascend iv, advanced system for computations in engineering design. Technical report, Department of Chemical Engineering, Carnegie Mellon University, june 1998.
[157] May Wolfang. Florid: User Manual.
[158] D. R. Wood. An algorithm for finding a maximum clique in a graph. Operations Research Letters, 21:211-217, January 1997.
[159] Guizhen Yang, Michael Kifer, Hui Wan, and Chang Zhao. Flora-2: User's Manual.

\section*{Appendix A}

\section*{Frame Logic \(-\mathcal{F}_{\text {LoRA- }}\) 2}

\begin{abstract}
Frame Logic (FL) [102] is a formalism that provides higher expressive power, in order to represent entities, relations and processes between these entities. At the same time, since the main concept of FL is represented as an object, FL embraces many aspects of ObjectOriented languages, expressed in a richer way. FL sees the objects as active entities (and not just as data structures with simple process attached), overcoming the deficiencies of the classic Object-Oriented paradigms to represent knowledge and reasoning. There are many (partial) implementation of FL, like FLORID [157], Ontoprise [10] and FLORA-2 [159]. We will focus on the \(\mathcal{F L O R A}\)-2 variant, since it was the one selected to develop this thesis.
\end{abstract}

\section*{A. 1 FL Syntax}

Fl allows the representation of two expression classes: terms and formulas, which can be constructed from the following sets of symbols

\section*{A.1.1 Alphabet}

Variable symbols: the variables will be represented by finite sequences of characters, having the following restriction: they must start with the symbol ?, followed by 0 or more letters and/or digits and/or the character _.

For instance: ?Q, ?warehouse_12, ?_, ?_num.
Constant symbols: There three syntactically different types of constants: symbolic constants denoted by finite sequences of alpha-numeric characters plus _ (not starting with ?); numeric constants represented according to the usual arithmetic rules; and ASCII character strings being any symbol combination enclosed in simple quotes.

For example: a, Warehouse_01, 1, 25, 35.0, 's89_54h', 'Q\#G_T_'.
First-order function symbols: These symbols have a name and an arity. The name is symbolic constant and the arity is a numeric constant \(n \geq 0\). Constants can be seen as a functional symbol of arity 0 .

For instance: \(f / 2\), author_of \(/ 1\).

\section*{Constructors:}
- The constants are constructors of arity 0 ,
- The following characters are used in class construction : [, ], \{, \}, ",", ";", \%, ->, *->, =>, *=>, :, ::

Predicate symbols: These symbols have also a name and an arity. The name is a symbolic constant and the arity is a number. The following examples present a general notation: \(\mathrm{p} / 1\), ancestor_of \(/ 2\).

Note there is no syntactical difference between functional and predicate symbols.
Logic operators:
- Negation: + classic Prolog negation as failure, not negation implementing well-founded semantics [159].
- Disjunction: ";"
- Conjunction: ","
- Implication: ":-"

Auxiliary symbols: Grouping symbols, "(", ")".

\section*{A.1. 2 Terms}

The class of first-order terms (FoT) of FL es the smallest class that satisfies the following requirements. Let \(t\) be a linear finite sequence of symbols, then:
1. If \(t\) is a variable, then \(t \in\) FoT,
2. If \(f_{m} / 0\) is a function, then \(f_{m} / 0 \in\) FoT,
3. If \(f_{m} / n\) is a function and \(t_{1}, \ldots, t_{n} \in \operatorname{FoT}\), then \(f_{m}\left(t_{1}, \ldots, t_{n}\right) \in\) FoT.

FoTs will be used to name objects, methods and classes. Therefore they are called identifier terms (term-id). Between the FoTs we can point out the basic FoTs (i.e. a FoT with no occurrence of variables). The basic FoTs are used to identify objects (objeto-id). For example, ?x, warehouse, 'Ana', siblings (?Parent), age_of( siblings (?Parent)), are FoTs.

On the other hand, the class of higher-order terms (HoT), is the one ensuring the following requirements:
1. If \(t \in\) FoT, then \(t \in\) HoT.
2. If \(t \in\) FoT and \(t_{1}, \ldots, t_{n} \in \operatorname{HoT}\), entonces \(t\left(t_{1}, \ldots, t_{n}\right) \in\) HoT.

For instance: \(a, f(? X), ? \times(s, ? x), ? \times(f, ? Y)(? x, g(k))\), are HoTs.
The HoTs allow the naming of (like the FoTs) objects, methods and classes. Moreover, HoTs permit to name predicates. Observe that the HoTs are built from a wider set of symbols (with respect to the FoTs), enabling the usage of variables or object schemes while naming a function.

\section*{A.1.3 Formulas}

In order to describe the objects, FL uses formulas, formed of conjunctions and/or disjunctions: of atomic formulas in an extended representation and/or molecules in a compact representation. First we will see the more elementary (or atomic) object descriptions, called data atoms. In a second place, we will introduce the data atoms that can be built using the the constructors previously introduced.

The symbols \(O, M, V\) and \(T\) are called meta-variables, which represent any term that names an object, a method or a value, depending on the place they take in the following descriptions. The comments introduced are self explanatory.

\section*{Data atoms:}
1. \(\mathrm{O}[\mathrm{M}->\mathrm{V}]-\operatorname{method} \mathrm{M}\) applied to object O provides the result V ;
2. \(\mathrm{O}\left[\mathrm{M}-\gg\left\{\mathrm{V}_{1}, \ldots, \mathrm{~V}_{n}\right\}\right]-\operatorname{method} \mathrm{M}\) applied to object O provides as a result the set \(\mathrm{V}_{1}, \ldots, \mathrm{~V}_{n}\).

We can point out a particular type of formula to represent objects, called molecule. In fact, a molecule is a formula that represents an object in a more compact way.

\section*{Molecules:}
1. A formula of the form:
\(\mathrm{O}\left[\mathrm{M}_{1}\right.\) Conss \(\left._{1} \mathrm{~V}_{1}\right], \ldots, \mathrm{O}\left[\mathrm{M}_{n}\right.\) Cons \(\left._{n} \mathrm{~V}_{n}\right]\), with Cons \({ }_{i} \in\{->,-\gg\}\), can be denoted in a compact way as, \(\mathrm{O}\left[\mathrm{M}_{1}\right.\) Cons \(_{1} \mathrm{~V}_{1}, \ldots, \mathrm{M}_{n}\) Cons \(\left._{n} \mathrm{~V}_{n}\right]\)
2. A formula of the form:
\(\mathrm{O}\left[\mathrm{M}_{1}\right.\) Cons \(\left._{1} \mathrm{~V}_{1}\right] ; \ldots ; \mathrm{O}\left[\mathrm{M}_{n}\right.\) Cons \(\left._{n} \mathrm{~V}_{n}\right]\), with Cons \(_{i} \in\{->,-\gg\}\),
can be denoted in a compact way as, \(\mathrm{O}\left[\mathrm{M}_{1}\right.\) Cons \(_{1} \mathrm{~V}_{1} ; \ldots ; \mathrm{M}_{n}\) Cons \(\left._{n} \mathrm{~V}_{n}\right]\)
3. A formula of the form:
\(\mathrm{O}\left[\mathrm{M}_{1}\right.\) Cons \(\left._{1} \mathrm{~V}_{1}\right], \mathrm{V}_{1}\left[\mathrm{M}_{2}\right.\) Cons \(\left._{2} \mathrm{~V}_{2}\right], \ldots, \mathrm{V}_{n-1}\left[\mathrm{M}_{n}\right.\) Cons \(\left._{n} \mathrm{~V}_{n}\right]\)
can be denoted in a compact way as,
\(\mathrm{O}\left[\mathrm{M}_{1}\right.\) Cons \(_{1} \mathrm{~V}_{1}\left[\mathrm{M}_{2}\right.\) Cons \(_{2} \mathrm{~V}_{2}\left[\ldots \mathrm{~V}_{n-1}\left[\mathrm{M}_{n}\right.\right.\) Cons \(\left.\left.\left.\left._{n} \mathrm{~V}_{n}\right] \ldots\right]\right]\right]\)
4. A formula of the form:

O:C, O[M Cons V] can be denoted in a compact way as, O:C[M Cons V] or O[M Cons V]:C

\section*{Type atoms}

The Type atoms describe the type T provided by a method M when applied to a class C . Type atoms are also called signatures.

\section*{Type atoms (signatures)}
1. \(C[M=>T]-\operatorname{method} M\) is applied to class \(C\) returning a value of type \(T\),
2. \(C[M=\gg T]-\operatorname{method} M\) is applied to class \(C\) returning a set of values of type \(T\).

For example,
```

person[name => -string].

```
paper[authors \(\Rightarrow>\)-string].

The signature molecules are defined using the same methodology of data molecules. For instance,
```

person[name => _string, siblings(person) =>> person].
paper[authors }=>>> person, title => _string]

```

\section*{Classes}

A type molecule defines partially a class. Therefore, the previously exposed examples are class partial definitions, person and paper.

Class can be seen as the abstraction of a group of objects, or as the type of such objects. For example,
denotes that Jane is a person, and alternatively that Jane is a value of type persona.
In FL we can define the inheritance relation using the operator ::,
```

woman:: person.

```
this way we are specifying that the class woman is a subclass of person.
Furthermore, using the operators \(*->, *-\gg, *=>\) and \(*=\gg\), we can define inheritable properties and signatures, for example,
```

person[name *=> _string, siblings(person) *=>> person, mother *=> woman].

```

Having the previous definitions, the following rules define the concept of formula, where \(\mathcal{F}\) is a metavariable.
1. Let \(P / n\) be a n-ary predicate symbol and \(T_{1} \ldots T_{n}\) be terms, then \(P\left(T_{1}, \ldots, T_{n}\right)\) is a formula;
2. Data atoms, molecules and predicates are formulas, signature atoms are also formulas;
3. If \(\mathcal{F}\) is a formula, then not \(\mathcal{F}\) is a formula;
4. If \(\mathcal{F}_{1}\) and \(\mathcal{F}_{2}\) are formulas, then \(\left(\mathcal{F}_{1}, \mathcal{F}_{2}\right)\) and \(\left(\mathcal{F}_{1} ; \mathcal{F}_{2}\right)\) are formulas;
5. If \(\mathcal{F}_{1}\) is a molecule and \(\mathcal{F}_{2}\) is a formula, then \(\mathcal{F}_{1}:-\mathcal{F}_{2}\) is a formula.

\section*{Some method examples}

The methods can be defined not just as properties, but also through rules. This rules determine, in a dynamic way, the result of the method in question.
```

?p[age -> ?a] :- ?p:person,
?p[dob[year -> ?year]],
present_date[year -> ?pyear],
?a is ?pyear - ?year.

```

Note that this rule is applied each time a goal ? \(\times\) [age -> ?a] appears. This goal causes the evaluation of the rule only if ?x:person (the first thing verified in the rule). Therefore, this is a rule that is only applied to objects of type person.

\section*{Virtual classes}

Virtual classes are classes which compute their members in a dynamic way, using rules. These rules evaluate whether a certain object belongs or not to the class in question. For instance,
?x: tall_person :- ? x: persona, ? [height \(\rightarrow\) ? h], ?h \(>1.80\).
As can be seen, this example defines the class of tall_person s , which is formed from the persons with height grater that 1.80.

\section*{A.1.4 Higher-Order Logic}

Higher-Order Logic (HOL), in FL, allows us to represent functor terms (including object-id) and predicates. In HOL, complex terms (HoT) may appear wherever function term can appear, for example, group \((? x)(? y, ? z)\) is a HoT, that belongs to HOL, where the function symbol is the term group \((? x)\). Variables in HOL can take values on the set of terms, predicates, functions and even atomic formulas.

Using this potentiality we can, for instance, implement an algorithm capable of computing the transitive closure of any transitive binary relation.
```

tclosure(?p)(?x, ?y) :- ?p(?x, ?y).
tclosure(?p)(?x, ?y) :- ?p(?x, ?z), tclosure(?p)(?z, ?y).

```

Note that this predicate is parametrized using the name of the relation of which we want to find the closure. Also, observe that this predicate is highly prone to go into an infinite loop if the relation has some symmetry. This problem is avoided in \(\mathcal{F}\) LORA-2, due to the table mechanism [159], which stores the previously computed terms, saving the system from processing them again.```

