

Transforming Mathematical Models Using Declarative Reformulation Rules

Antonio Frangioni

Dipartimento di Informatica, Università di Pisa
Polo Universitario della Spezia, Via dei Colli 90, 19121 La Spezia, Italy
`frangio@di.unipi.it`

Luis Perez Sanchez

Dipartimento di Informatica, Università di Pisa
Largo B. Pontecorvo 3, 56127 Pisa, Italy
`perez@di.unipi.it`

Abstract

Reformulation is one of the most useful and widespread activities in mathematical modeling, in that finding a “good” formulation is a fundamental step in being able to solve a given problem. Currently, this is almost exclusively a human activity, with next to no support from modeling and solution tools. In this paper we show how the reformulation system defined in [15] allows to automatize the task of exploring the formulation space of a problem, using a specific example (the Hyperplane Clustering Problem). This nonlinear problem admits a large number of both linear and nonlinear formulations, which can all be generated by defining a relatively small set of general Atomic Reformulation Rules (ARR). These rules are not problem-specific, and could be used to reformulate many other problems, thus showing that a general-purpose reformulation system based on the ideas developed in [15] could be feasible.

Key Words: Automatic Reformulation, Mixed-Integer NonLinear Programs, Declarative Techniques

1 Introduction

It is a striking discovery that while the term *reformulation* is ubiquitous in mathematics (e.g. [4, 9, 16, 18]), there are few formal definitions and theoretical characterizations of the concept. 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 [11]. These reformulations are capable of exploiting *syntactical structure* of the model, such as presence of particular algebraic terms in parts of its algebraic description [7]. While being very relevant, these do not include all transformations that have shown to be of practical use.

Indeed, 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 [1]) and the equivalence between the optimal solution value of a convex problem and that of its dual. 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 those 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 [17] 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 [2], 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 [6] or even exponential in theory [3, 5, 8], but quite effective in practice. Limited to MIP problems, general ideas based on variable redefinition were proposed by [13, 14], without finding wide application due to its complexity, remaining unknown (or unused) by the “average” user. 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 [11, 12].

However, a general formal definition of reformulation is not enough; 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.

In this paper we showcase the modeling capabilities of the I-DARE (Intelligence-Driven Automatic Reformulation Engine) system developed in [15] by using a specific example (the Hyperplane Clustering Problem). This nonlinear problem admits a large number of both linear and nonlinear formulations, which can all be generated by defining a relatively small set of general *Atomic Reformulation Rules* (ARR) on a set of properly defined *structures* described in §2.

The ARRs are a key component of the I-DARE reformulation system (I-DARE(**t**)) [15]; it informally defines a reformulation rule based on the fact that we can transform structure *A* into *B* if and only if *A*’s input is transformable into *B*’s input, and *B*’s output is transformable into *A*’s output. ARRs are defined between two structures; in the I-DARE system, structures are classes that are derived from the hierarchy in Figure 1 where `d_LeafProblem_C` represents the atomic structures, and `d_Block_C` represents the structures that are composed of other structures. The composition of structures is controlled by the arrangement of the sub-structure’s shared variables. I-DARE exploits the power of a declarative language (in particular *FLORA-2* [19]) for the definition of the structures and of the ARRs.

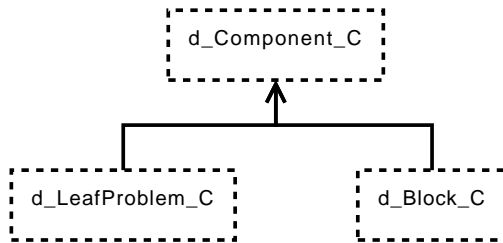


Figure 1: I-DARE(lib) hierarchy

ARRs are divided in two classes, Algebraic ARRs (ARR^Σ) and Algorithmic ARRs (ARR^A). The ARR^Σ s define the transformation of the input and output using solely algebraic operation, whereas the ARR^A s need the intervention of an algorithmic approach for reformulation the input and/or output. In this paper, for space reasons, we only concentrate on the former. Further, we will not define formally the concept of ARR, which is described in detail in [15]. The aim here is to show that a relatively small set of general (algebraic) ARRs suffice for producing a large number of both linear and nonlinear formulations for the problem. These ARRs are not problem-specific, and could be used to reformulate many other problems, thus showing that a general-purpose reformulation system based on the ideas developed in [15] could be feasible.

2 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)],
  rplR -> [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, bv],
  subsC -> [d_Component_C, d_SingleBV_C],
  link -> [(X, d_all), (Y, d_all)],
  rplR -> [ct = 1, bv = 1]
].

```

Considering operators like $|\cdot|$ (absolute value), we can create further structures. For instance the following leftmost structure represents $|\sum_i v_i c_i|$, where v_i is a CV and c_i is a constant, and the rightmost represents its non-vectorial version.

<pre> d_VAbs_C :: d_LeafProblem_C [dim_var -> [D], args -> [v = d_vector(d_var, [D]), c = d_vector(d_constant, [D])]]. </pre>	<hr/> <pre> d_SAbs_C :: d_LeafProblem_C [args -> [v = d_var, c = d_constant]]. </pre>
---	--

Structures representing specific collections of constraints and/or optimization problems can also be defined, like Linear Programs (`d_LP_C`); Mixed-Integer Linear Programs (`d_MILP_C`); Semi-Assignment Constraints (`d_SemiAssign_C`), and Complementary Constraints (`d_ProdCC_C`) defined by $xy = 0$ where $x, y \geq 0$ are CVs.

```

d_LP_C :: d_LeafProblem_C
[
  dim_var -> [cols, cons],
  args -> [
    x = d_vector(d_var, [cols]),

```

```

    c   = d_vector(d_constant, [cols]),
    A   = d_vector(d_constant, [cons, cols]),
    b   = d_vector(d_constant, [cons]),
    rels = d_vector(d_rel, [cons]),
    dir = d_direction
  ]
].

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

d_SemiAssign_C :: d_LeafProblem_C
[
  dim_var -> [D],
  args -> [
    v = d_vector(d_var, [D])
  ]
].

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 $f \leq/\geq/= c$, where c is a constant, and f can be any component. Likewise we could define a minimization objective function,

```

d_Constraint_C :: d_Block_C
[
  ids -> [expr, rel, c],
  subsC -> [d_Component_C, d_Relation_C, d_Constant_C],
  link -> [[X], d_all], ([], d_all), ([], d_all)],
  rplR -> [expr=1, rel = 1, c = 1]
].

d_OFMin_C :: d_Block_C
[
  ids -> [expr],
  subsC -> [d_Component_C],
  link -> [[X], d_all],
  rplR -> [expr = 1]
].

```

Note that in `d_Constraint_C`, `d_Relation_C` and `d_Constant_C` are helper structures to put a single relation and/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 (also in the case of `d_OFMin_C`).

Once we have the single structures we may want to compose them to obtain more complex structures. The following structure combines two structures that share a set of variables,

```

d_Composition_C :: d_Block_C
[
  ids -> [p1, p2],
  subsC -> [d_Component_C, d_Component_C],
  link -> [[X,Y], d_all], ([X,Z], d_all)],
  rplR -> [p1 = 1, p2 = 1]
].

```

Observe that both substructures share a set of variables (x) and have independent sub-sets 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   -> [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 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.

3 Creating a model

In this section we propose the representation of a Hyperplane Clustering Problem (HCP) using an I-DARE model. In a HCP we have a set of points $p = \{p_i \mid i \in M\} \in \mathbb{R}^D$ and we want to find the set of N hyperplanes $w = \{w_{j1}x_1 + \dots + w_{jd}x_d = w_j^0 \mid j \in N\} \in \mathbb{R}^D$ 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,

$$\min \sum_{i \in M} \sum_{j \in N} |w_j p_i - w_j^0| x_{ij} \quad (3.1)$$

$$\sum_{j \in N} x_{ij} = 1 \quad \forall i \in M \quad (3.2)$$

$$\sum_{k \in D} |w_{jk}| = 1 \quad \forall j \in N \quad (3.3)$$

$$w \in \mathbb{R}^{N \times D}, \quad w^0 \in \mathbb{R}^N, \quad x \in \{0, 1\}^{M \times N}$$

Note HCP has a parameter $p \in \mathbb{R}^{M \times D}$, and dimensions $N, M, D \subset \mathbb{N}$.

To model HCP we will use a combination of the previously specified structures. Note that (3.1) is an objective function containing products between absolute values and BVs; (3.2) is a semi-assignment; and (3.3) is a constraint containing absolute value operations. Hence, we can build the following model.

Dimensions, indices and Properties

<pre> d_dimension(D). d_dimension(N). d_dimension(M). d_index(i, M). d_index(j, N). d_index(k, D). p : d_constant . p : d_property [dims -> [M, D]]. w : d_var . w : d_property [dims -> [N, D]]. </pre>	<hr/> <pre> w0 : d_var . w0 : d_property [dims -> [D]]. x : d_var . x : d_property [dims -> [M, N], lower -> 0, upper -> 1]. </pre> <hr/>
--	--

Structures

```

vabsof : d_VAbs_C
[
  args -> [
    v = $(w(j,k),[k]), w0(j)),
    c = $(p(i,k),[k], 1)
  ] //freeinds i,j
].

bvof : d_SingleBV_C
[
  args -> [
    v = x(i,j)
  ] //freeinds i,j
].

semiof : d_SemiContinuous_C
[
  subs -> [absof, bvof],
  subVP -> [[(w,w0)], [x]],
  freel -> [i,j]
].

indof : d_IndComposition_C
[
  subs -> [semiof],
  subVP -> [[(w,w0,x)]]
].

of : d_OFMin_C
[
  subs -> [indof],
  subVP -> [[(w,w0,x)]]
].

semiac : d_SemiAssign_C
[
  args -> [
    v = $(x(i,j),[j])
  ] //freeinds i
].

sabsc : d_SABs_C
[
  args -> [
    v = w(j,k),
    c = 1,
  ] //freeinds j,k
].

rel : d_Relation_C
[
  args -> [rel = '=']
].

c : d_Constant_C
[
  args -> [c = 1]
].

indc1 : d_IndComposition_C
[
  subs -> [sabsc],
  subVP -> [[(w)], [j]],
  freel -> [j]
].

constraint : d_Constraint_C
[
  subs -> [indc1, rel, c],
  subVP -> [[w], [[]], [[]]],
  freel -> [j]
].

indc2 : d_IndComposition_C
[
  subs -> [semiac],
  subVP -> [[(x)]]
].

indc3 : d_IndComposition_C
[
  subs -> [constraint],
  subVP -> [[(w)]]
].

cmpdc : d_Composition_C
[
  subs -> [indc2, indc3],
  subVP -> [[[]], [x], [[]], [w]]
].

fcmp : d_Composition_C
[
  subs -> [of, cmpdc],
  subVP -> [[(x,w), w0], [(x,w), []]]
].

HCP : d_Formulation
[
  root -> fcmp,
  dimensions -> [D,M,N],
  indices -> [i,j,k],
  properties -> [w,w0,p,x]
].

```

The diagram in Figure 2 shows the HCP formulation by representing only the name and class of the structures used, plus the relations between them:

4 Reformulations

In this section we will introduce some of the reformulations that can be created based on the previously defined structures. One usual goal when reformulating nonlinear problems is to remove the nonlinear elements, e.g. by adding the proper additional variables and constraints. We will use the classes `d_MILP_C` and `d_LP_C` as the main goals in the ARR to be presented herein. Most of the reformulation rules exposed in this section were extracted from [10].

In some of the cases, the generated MILP and LP will have no objective function (i.e. the cost is constant), 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 produce, we will transform the cost constants of the objective function.

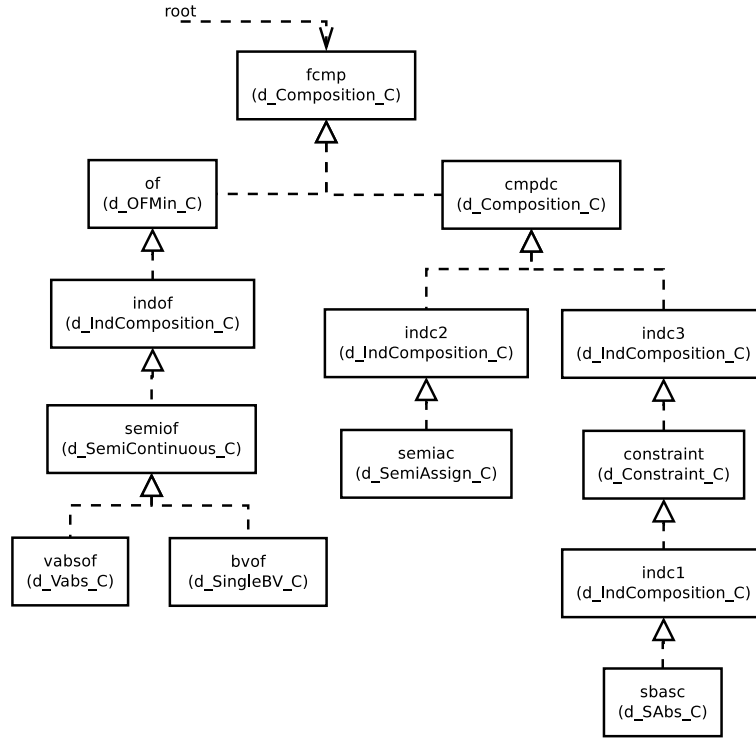


Figure 2: HCP Formulation

4.1 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: $w - Ub \leq 0$, $w - x \leq 0$ and $x + Ub - w \leq U$. So we can build the following ARR^Σ .

```

d_ProdBC_to_MILP_ARR : d_ARR_Algebraic
[
  A -> d_ProdBC_C(?, ?),
  B -> d_MILP_C,
  indexA -> [1=(i, i1), 2=j],
  indexB -> [],
  dimRel -> [cols1=1, colsR=2, cons=3],
  arg_map -> [
    B..ci = 0,
    B..cR = $([0, 1]),
    B..Ai = $([
      [ $(-1*_up(A..cont..v), [i1, i]) ],
      [ $ ( 0, [i1, i]) ],
      [ $ ( _up(A..cont..v) , [i1, i]) ]
    ]),
    B..Ar = $([
      [ $_cs([1->j=1, 0]), [i1, j] ] ],
      [ $_cs([1->j=1, -1]), [i1, 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]
  ],
  ans_map -> [
    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 bx , and the rest of the constants are 0. The utility of this objective function constants will be seen when reformulating d_OFMin_C and d_Constraint_C . Most of the ARR's proposed in the rest of the section are written in a similar way to the one previously exposed. Therefore, we will avoid the specification of the ARR subclass (in I-DARE(τ) language) due to space limitations (for the complete set of ARR consult [15]).

4.2 SAbs to Composition

If we consider a structure involving a term $|pv|$ (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 $|pv|$ by $t^+ + t^-$; and adding the constraints $pv - t^+ - t^- = 0$ and $t^+t^- = 0$. This reformulation involves a linear substructure, plus a complementary constraint ($xy = 0$). So we can define an ARR^Σ that transforms d_SAbs into a composition between a d_LP_C and a d_ProdCC_C . Notice that the substitution of $|pv|$ may be expressed by defining the constants in d_LP_C with 0 for v and 1 for t^+ and t^- .

4.3 VAbs to LP

Considering now a term $|\sum_i p_i v_i|$ 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. In this case we used a condition inside the ARR^Σ indicating that A must have a parent d_OFMin_C inside the block's tree, thus it must be inside a minimization function.

4.4 SemiContinuous to MILP

If we manage to narrow a $\text{d_SemiContinuous_C}$ until the point of knowing that it has an d_LP_C inside, then we can easily transform $\text{d_SemiContinuous_C}$ into a d_MILP_C . Assume the LP has the form (leftmost equation)

$$\begin{array}{lll}
 \min c^T x & \min \sum_i c_i x_i y & \min \sum_i w_i \\
 Ax = b & Ax = b & Ax = b \\
 x_i \in [0, B_i] & x_i \in [0, B_i], \quad y \in \{0, 1\} & w_i - B_i y \leq 0 \quad \forall(i) \\
 & & w_i - x_i \leq 0 \quad \forall(i) \\
 & & x_i + B_i y - w_i \leq B_i \quad \forall(i) \\
 & & w_i, x_i \in [0, B_i], \quad y \in \{0, 1\}
 \end{array}$$

then the fact of multiplying this LP by a BV y (only in the objective function) creates the following MINLP (previous center equation). This MINLP can be reformulated into a MILP by applying the same mechanism used for $\text{d_ProdBC_to_MILP_ARR}$ (cf. §4.1). We may add a CV $w_i \in [0, B_i]$ 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 MILP present in the rightmost part of the previous equations.

4.5 ProdCC to MILP

When in presence of a complementary constraint $xy = 0$, we can substitute it by the following MILP constraints, $x - Mz \leq 0$ and $y + Mz \leq M$, where $z \in \{0, 1\}$ and M is a sufficiently large number. Since d_ProdCC_C represents a constraint, the generated MILP will have no objective function.

4.6 SemiAssign to MILP

The semi-assignment constraint $\sum_i y_i = 1$, has trivial transformation into a MILP with no CVs.

4.7 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 ($\sum_i c_i x_i$) 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 $d_constant$. Notice that an ARR^Σ to reformulate $d_Constraint_C(d_LP_C, \dots, \dots)$ into d_LP_C can be created in an analogous way.

4.8 OFMin to MILP

The reformulation of a d_OFMin_C with the inner structure narrowed to a MILP is even more direct than 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. Again in this case the reformulation from $d_OFMin(d_LP_C)$ to d_LP_C can be done in an analogous way.

4.9 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 $MILP^{j'}$, with $j \neq j'$. Therefore the resulting MILP can be composed as shown in Figure 3.

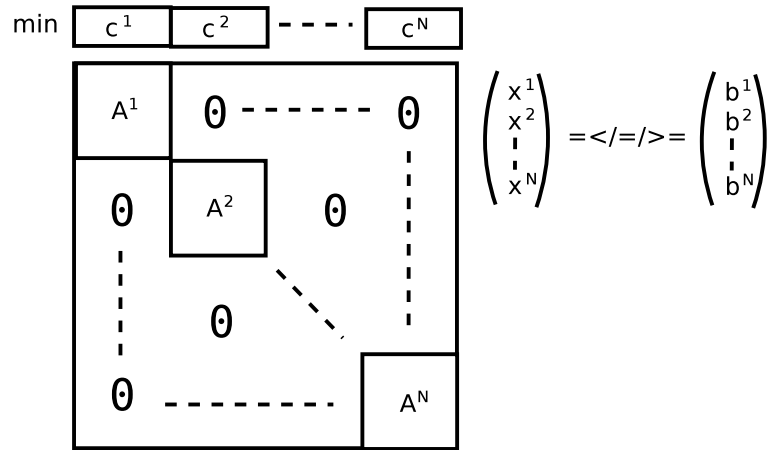


Figure 3: Independent Composition of N MILP subproblems

The c^j constants will be multiplied by the direction of $MILP^j$ in order to unify the objective function to a minimization. Applying this composition we can define the following ARR^Σ to reformulate a $d_IndComposition_C(d_MILP_C)$ into a single d_MILP_C . We could define a similar reformulation to integrate several d_LP_C into a single d_LP_C .

4.10 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 $\min(d^1 c_x^1 + d^2 c_x^2)x + c_y^1 y + c_z^2 z$, 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 MILP¹ and MILP², respectively;
- c_x^1 and c_x^2 are the costs related with the shared variables of MILP¹ and MILP², respectively;
- c_y^1 and c_z^2 are the costs related with the independent variables of MILP¹ and MILP², respectively;
- A_x^1 and A_x^2 are the LHS matrices related with the shared variables of MILP¹ and MILP², respectively;
- A_y^1 and A_z^2 are the LHS matrices related with the independent variables of MILP¹ and MILP², respectively; and
- b^1 and b^2 are the RHS vectors of MILP¹ and MILP², respectively.

The diagram in Figure 4 is a representation of this composition.

$$\min \begin{array}{|c|c|c|} \hline c_x^1 + c_x^2 & c_y^1 & c_z^2 \\ \hline \hline \hline \end{array} \begin{array}{|c|c|c|} \hline A_x^1 & A_y^1 & \mathbf{0} \\ \hline A_x^2 & \mathbf{0} & A_z^2 \\ \hline \hline \hline \end{array} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} b^1 \\ b^2 \end{pmatrix}$$

Figure 4: Composition of two MILP subproblems with shared variables

By using this integration mechanism we can define the ARR^Σ to reformulate $\text{d_Composition_C}(\text{d_MILP_C}, \text{d_MILP_C})$ into d_MILP_C . Other combinations of d_MILP_C and d_LP_C as substructures of d_Composition_C can conduct to similar ARR^Σ 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).

5 Applying the ARR^Σ s to HCP

Taking the HCP formulation we defined in §3, we could apply a combination of the previously defined ARR^Σ s until finally obtain a MILP formulation. To show how the HCP formulation is modified by the application of the ARR^Σ 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 will start from the original HCP formulation (see Figure 5).

First we apply the $\text{ARR}^\Sigma_{\text{d_VAbs_to_LP_oncond_OFMin_ARR}}$ (§4.3) to the structure `vabsof` in the formulation (see Figure 6). 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 $\text{ARR}^\Sigma_{\text{d_SemiContinuous_LP_SingleBV_to_MILP_ARR}}$ (cf. §4.4), see Figure 7. 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 $\text{ARR}^\Sigma_{\text{d_IndComposition_MILP_to_MILP_ARR}}$ (cf. §4.9). 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 $\text{ARR}^\Sigma_{\text{d_OFMin_MILP_to_MILP_ARR}}$ (cf. §4.8), since `of` has `d_MILP_C` as its inner structure, see Figure 8.

Figure 9 moves to the constraints part, starting by reformulating `sabsc` using $\text{ARR}^\Sigma_{\text{d_SAbs_to_Composition_LP_ProdCC_ARR}}$ (cf. §4.2).

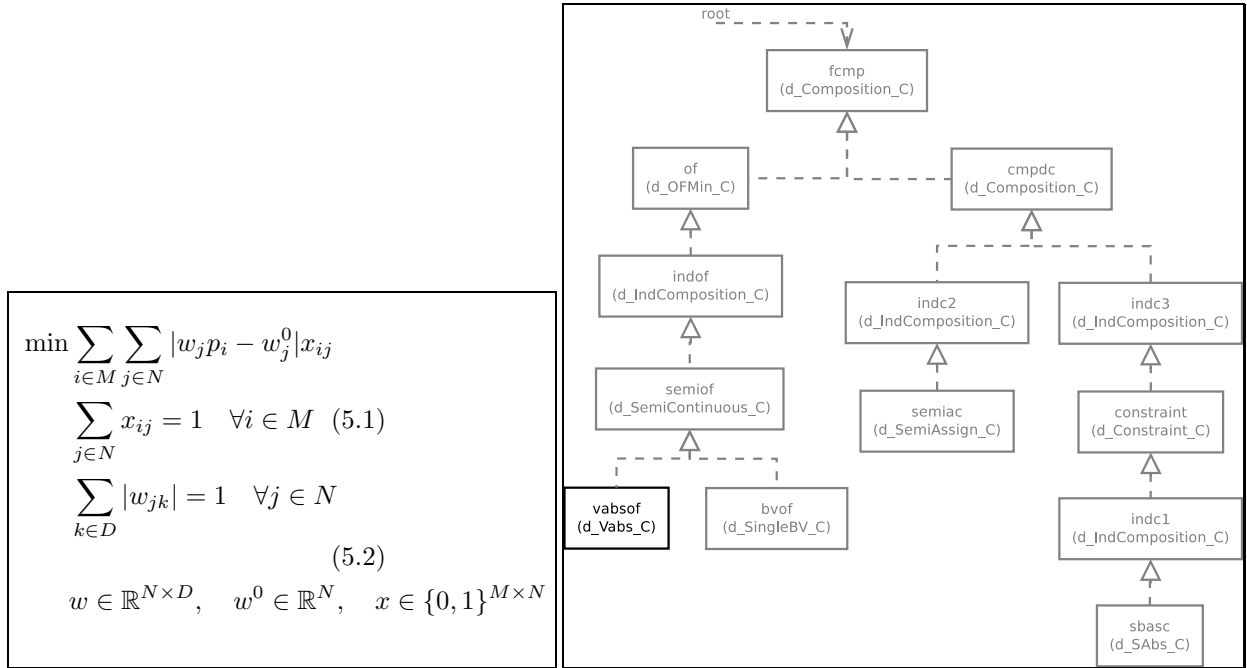


Figure 5: HCP non-linear formulation

Although in this case the complexity of the model augmented a little bit, this will allow us to simplify it further by applying $\text{ARR}_{\text{d_ProdCC_to_MILP_ARR}}$ (cf. §4.5) to the d_ProdCC_C structure class, that can be seen in Figure 10.

Observe that at this point the algebraic representation is in MILP form. However, the formulation still have to undergo some other reformulations to be completely transformed into a d_MILP_C , see Figure 11. Note how in this example the reformulations are applied only when the narrowing requisites are met. Only at that point the corresponding ARR_{Σ} can be applied to transform the structure. Thanks to the deductive power of $\mathcal{F}\text{LORA-2}$, the system easily detects which ARR s it can apply to a certain (maybe intermediate) formulation, allowing the creation of all possible reformulations.

6 Discussion

This paper shows, with a relatively simple example, how the I-DARE system allows to automatically produce a large set of reformulations of a given mathematical model based on a small set of general structures and Automatic Reformulation Rules. This system matches the capabilities of the framework envisioned in [11, 12], which covers a large number of real-life problems and reformulation techniques. However, our system also allows to deal with algorithmic reformulation rules that are out of reach for frameworks based exclusively on algebraic techniques, and it makes explicit use of the concept of *structure* to allow exploiting reformulation rules based on the semantic (as opposed to purely syntactic) meaning of each block. Our system also provides explicit algorithmic notions for its definition of reformulation, exploiting the power of declarative languages, unlike e.g. that of [17]. On the other hand, [2] manages the idea of mapping functions; while in theory it has the same power that our reformulation system has, we propose a reformulation system defined over a precise modeling language, that allows us to algorithmically and algebraically deduce reformulations. I-DARE(τ) 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.

As the example shows, a small set of structures and reformulation rules produces a large set of possible formulations. One of the main design goals of I-DARE(τ) is extensibility, i.e., the fact that one can easily define new structures and reformulation rules to cover all kinds of algebraic reformulations [11]. By doing so in a general way, i.e., defining reformulation rules for general models rather than for specific applications, the

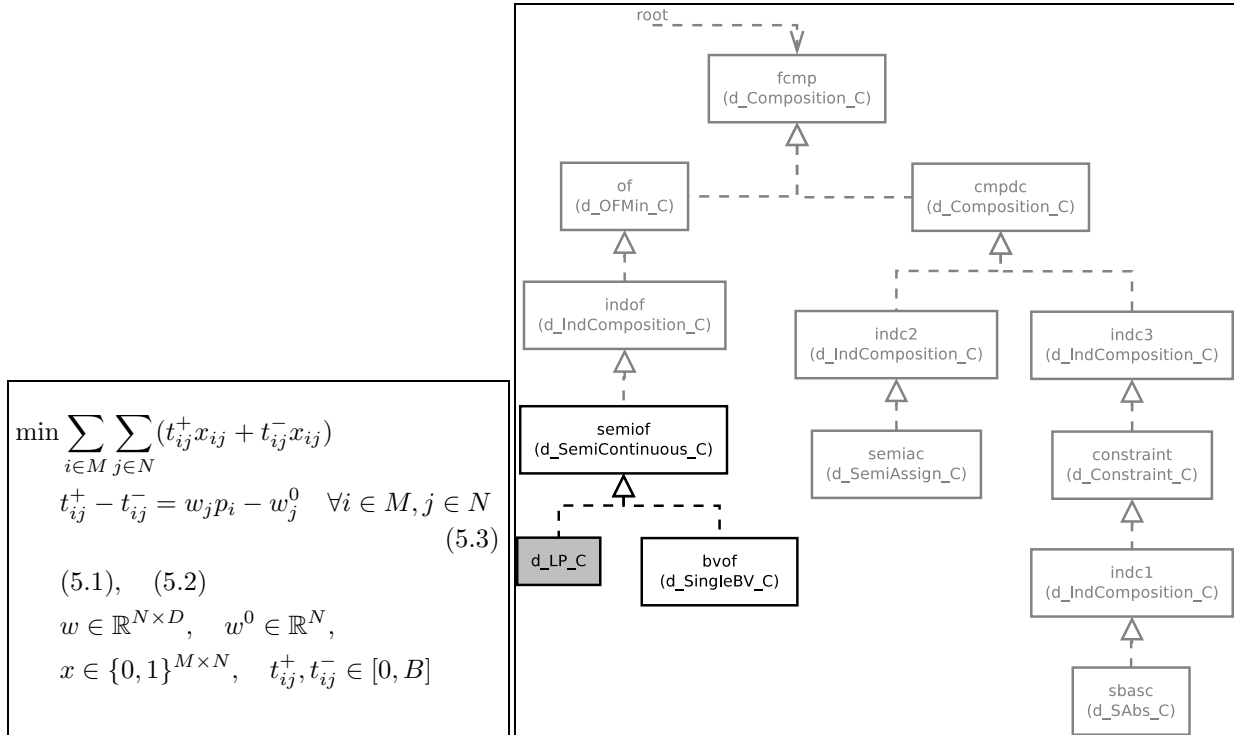


Figure 6: Transforming VABs to LP

system can then exploit reformulations developed for a specific model for entirely different classes of problems. This means that a system like I-DARE could act as a central repository for reformulation techniques, allowing more effective sharing of these ideas between researchers and practitioners and fostering a positive feedback loop whereby researchers in reformulation techniques find a much wider audience for their work, while practitioners have access to sophisticated reformulation techniques that they would be unlikely to develop (or even use) themselves. We believe that such a system could have a substantial positive impact both on the research in reformulation techniques and, possibly more importantly, on the practice of the solution of mathematical models.

References

- [1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993.
- [2] 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.
- [3] 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.
- [4] J. Bjorkqvist and T. Westerlund. Automated reformulation of disjunctive constraints in minlp optimization. *Computers and Chemical Engineering*, 23:S11–S14, 1999.
- [5] Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon, editors. *Column generation*. Springer, 2005.
- [6] A. Frangioni and B. Gendron. 0-1 Reformulations of the Multicommodity Capacitated Network Design Problem. *Discrete Applied Mathematics*, 157(6):1229–1241, 2009.

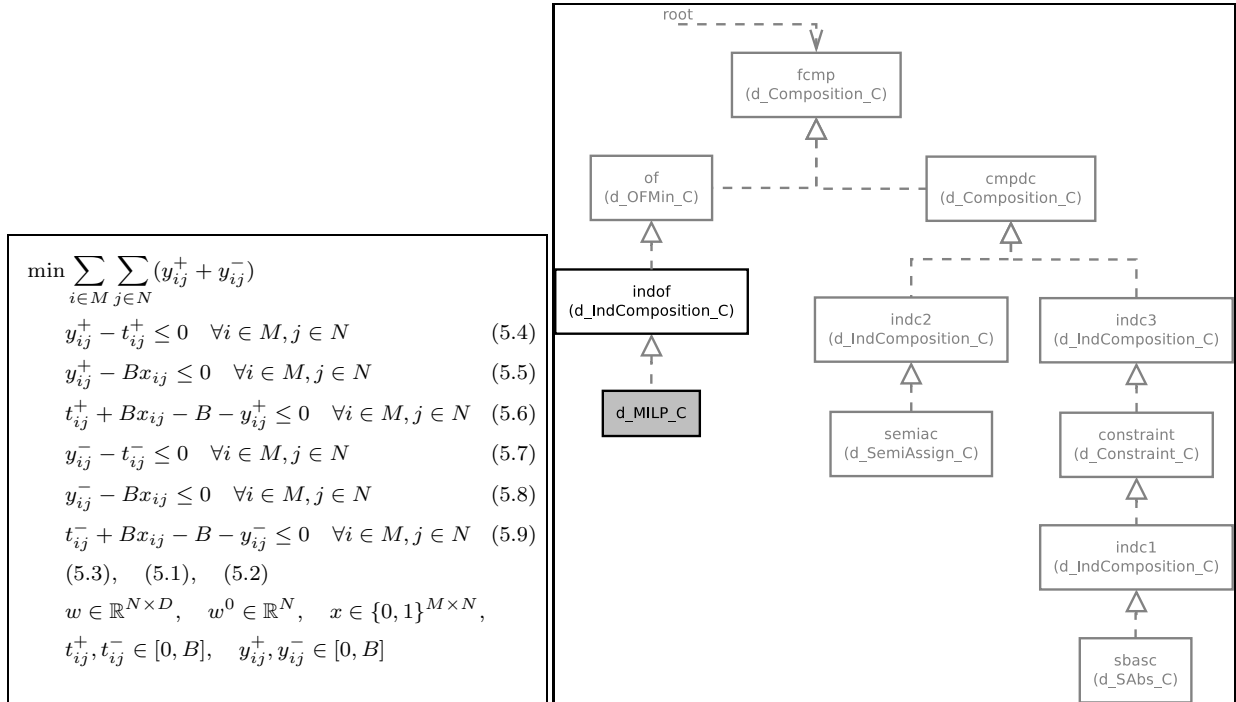


Figure 7: Transforming semi-continuous LP to MILP

- [7] A. Frangioni and C. Gentile. SDP Diagonalizations and Perspective Cuts for a Class of Nonseparable MIQP. *Operations Research Letters*, 35(2):181 – 185, 2007.
- [8] 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.
- [9] 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.
- [10] L. Liberti. Reformulation techniques in mathematical programming, in preparation. Thèse d’Habilitation à Diriger des Recherches, Université Paris IX.
- [11] L. Liberti. Reformulations in mathematical programming: Definitions and systematics. *RAIRO-RO*, 43(1):55–86, 2009.
- [12] L. Liberti, S. Caferi, 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.
- [13] R. Kipp Martin. Generating alternative mixed-integer programming models using variable redefinition. *Operations Research*, 35(6):820 – 831, 1987.
- [14] R. Kipp Martin. Using separation algorithms to generate mixed integer model reformulations. *Operations Research Letters*, 10(3):119 – 128, 1991.
- [15] Luis Perez Sanchez. *Artificial Intelligence Techniques for Automatic Reformulation and Solution of Structured Mathematical Models*. PhD thesis, University of Pisa, 2010.
- [16] .D. Serali and W.P. Adams. *A Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems*. Kluwer Academic Publishers, Dodrecht, 1999.

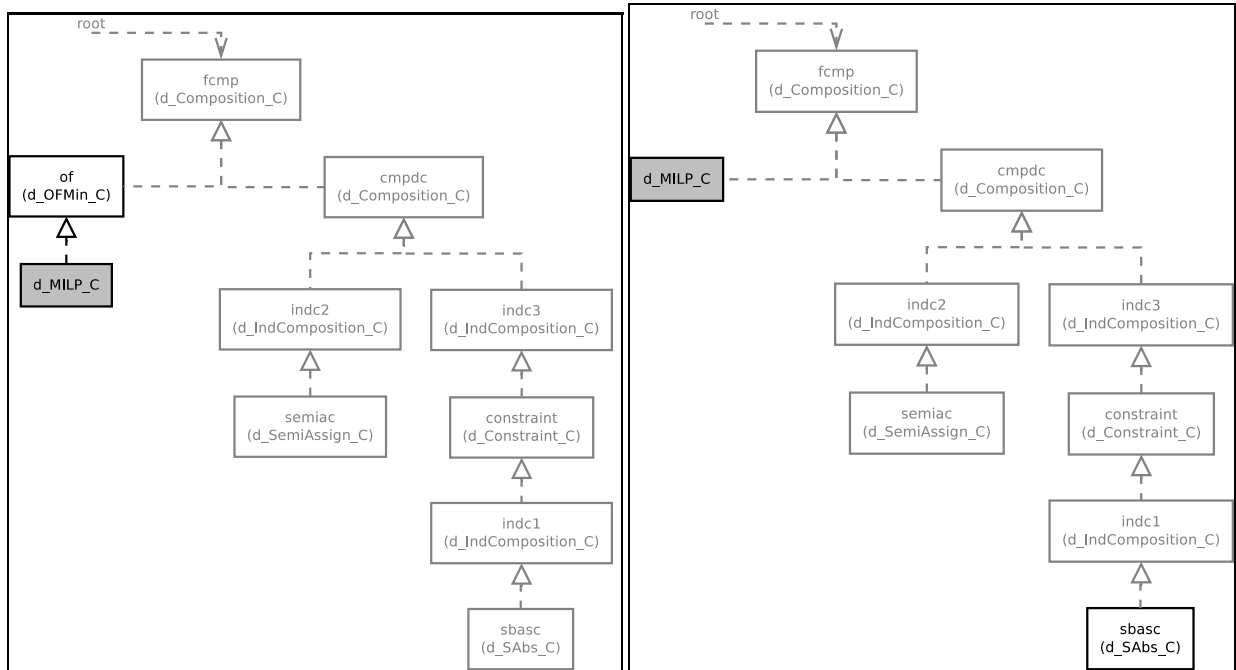


Figure 8: IndComposition to MILP and OFMin to MILP

[17] H. Serali. *Personal communication*. 2007.

[18] T.J. van Roy and L.A. Wolsey. Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35(1):45–57, 1987.

[19] Guizhen Yang, Michael Kifer, Hui Wan, and Chang Zhao. *Flora-2: User's Manual*.

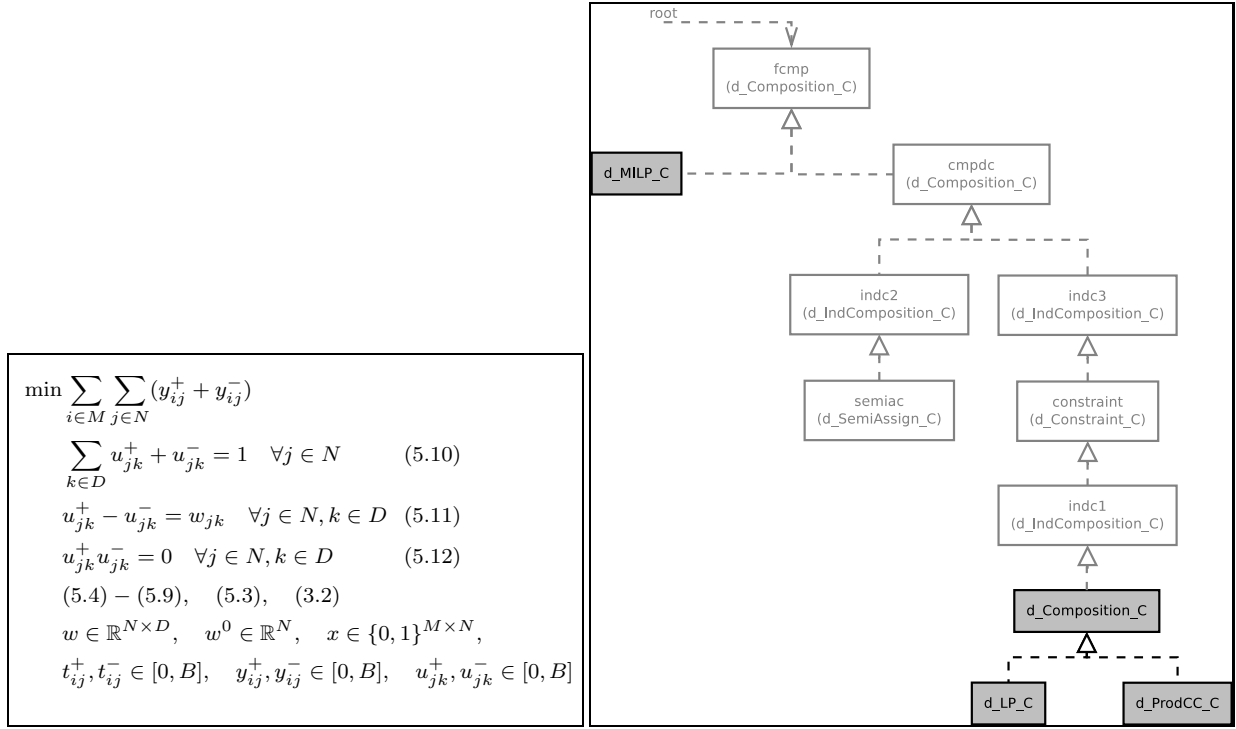


Figure 9: Transforming SAbS to Composition

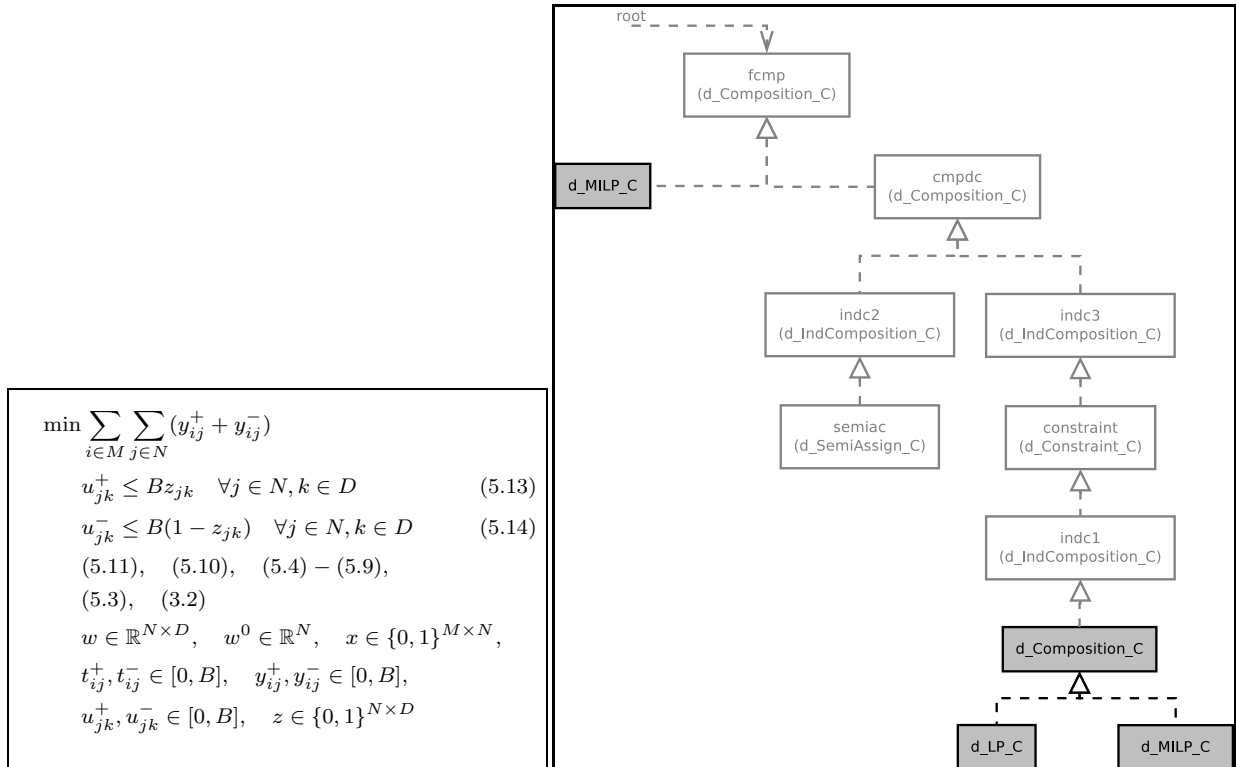


Figure 10: Transforming ProdCC to MILP

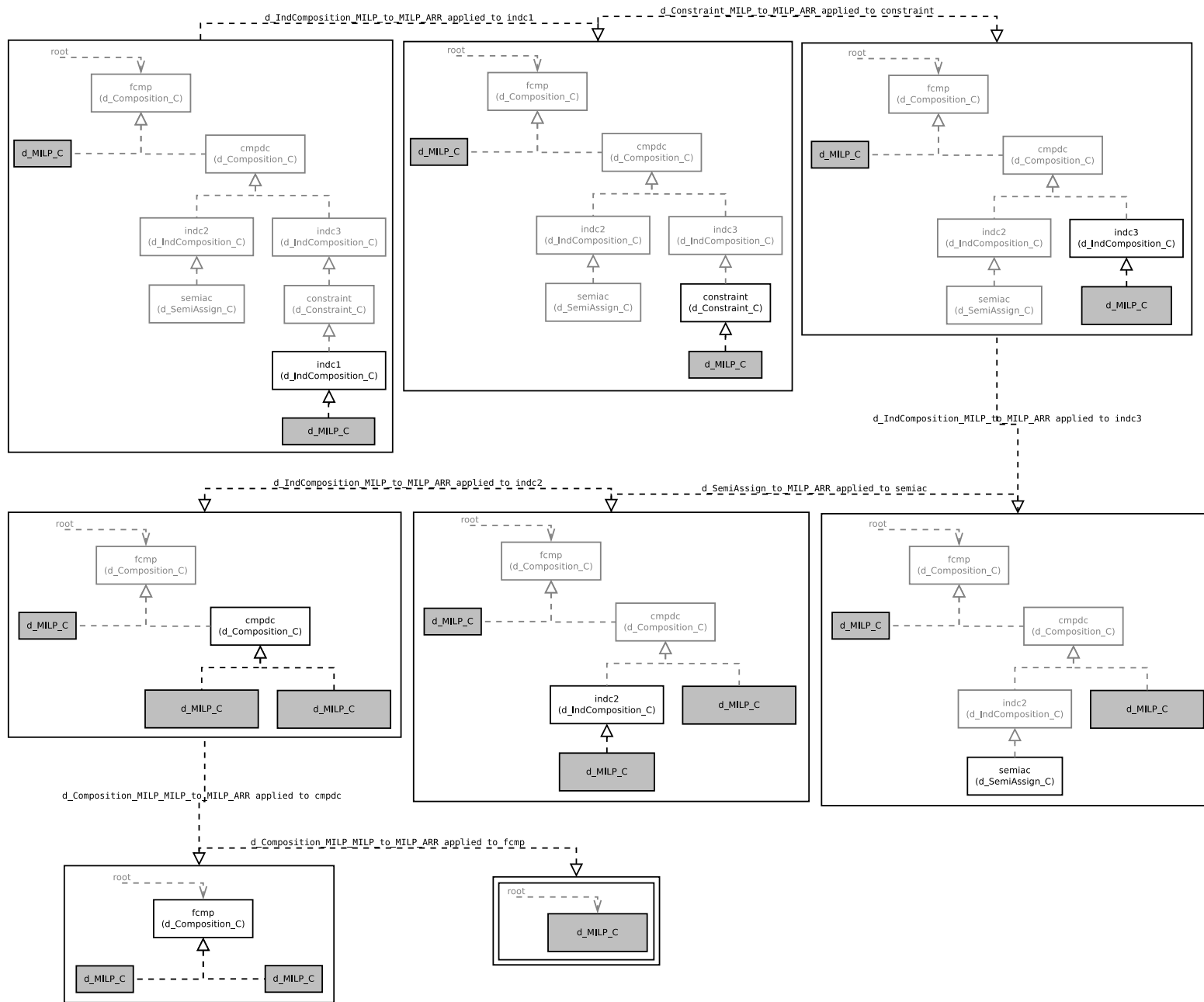


Figure 11: Rest of the reformulations