Mathematical Programming with DatalogLB

Release 0.2

LogicBlox

March 14, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

In this document we present BloxOptimize—the library extending LogicBlox with support for mathematical programming (MP). At this time BloxOptimize facilitates solving linear, mixed-integer programming (MIP) problems with limited forms of disjunction in the constraints. The library provides a set of syntactic constructs that allow specifying MP problems in this class using DatalogLB, an interpreter for these specifications, and an interface to several external MP solvers. The interpreter transforms problem specifications written by the users into the form readable by a selected solver. The interface then supplies that formulation to the solver, which performs required computations and returns the results. Finally, the interpreter writes the results into DatalogLB predicates. We introduce the BloxOptimize library by means of a set of test programs exploiting its capabilities. Our document consists of four parts:

- *Features*, which uses simple examples to demonstrate various problem features that BloxOptimize can handle,
- *Applications*, which describes more complex (and practical) applications of BloxOptimize,
- *Issues*, which lists some limitations of BloxOptimize,
- *Usage Notes*, which remarks on the use of the library, and finally,
- *Tutorial: Task Scheduling*, which provides a comperehsive presentation of the BloxOptimize library by means of its application to several, increasingly complex, variants of a real-life scheduling example.

All code samples shown in the document are directly cited from the BloxOptimize test suite directory `LogicBlox/BloxOptimize/testsuites/datalog-existential/` available in the current LogicBlox release.

CHAPTER

TWO

FEATURES

2.1 The Simplest Problem

The simplest case of MP problems are the problems in which the variables are the actual predicates. For example, consider the following problem with only one *variable*:

Maximize x with $0 \leq x \leq 3$

We begin our tutorial with encoding this simple problem.

The variable

Our implementation defines the variable as follows:

```
x[s]=v -> p(s),int[32](v), v<=3, v>=0.
p(s) -> x[s]=_.
lang:solver:variable('x).

p(s),p:id(s:n) -> string(n).
```

Above, the first clause declares *x* as a function with one key of a simple entity type and a value of a numeric type, and sets the lower and upper bounds on the value. The second clause is the existential constraint, not strictly necessary, but useful to check that the solver produces a solution. The third line states that the predicate *x* is a variable and that its values should be produced by invoking the solver. The fourth line declares the entity type for the key of *x*.

The objective function

The *objective function* is an numeric-valued function of the variable. From the Datalog point of view, it is an IDB predicate derived from the fact encoding the variable:

```
objective[]=z -> int[32](z).
objective[]+=x[_].
lang:solver:maximal('objective).
```

Using the familiar notation, the first two clauses provide the declaration and definition of the function predicate *objective*. Additionally, the third clause identifies the predicate symbol *objective* to represent the objective function for our program, and specifies the direction of the optimization (i.e., whether it is minimization or—as in this case—maximization).

The expected value

The program may also define the auxiliary EDB predicate representing the *expected value* of *x*, which will be used to test the correctness of the result:

```
x_expected[s]=y -> p(s),int[32](y).
```

The solver

Optionally, the following directive may be added to the program in order to specify which of the available external solvers should be used to compute the solution:

```
optimization:solverName[]="cbc".
```

If the solver directive is not given, default solver is used for the computation.

The above clauses form the *suite.program* block comprising a complete DatalogLB representation of our example MP problem.

The computation

To find the solution, we add the *suite.program* block to a workspace, and populate the EDB predicates by executing *test.init* block:

```
+p(s),+p:id(s:"s").
+x_expected[s] = 3 <- ^p(s).
```

Consequently, the predicate *x[s]* is populated with the value which solves the problem. The result is tested by the *test.assert* block:

```
+system:bloxunit:Compare:expected[c] = "x_expected",
+system:bloxunit:Compare:actual[c] = "x" <- system:bloxunit:Compare:id(c:0).
```

2.2 Including Constraints

In our introductory example, the only constraints were the bounds on the variable. This section shows how to specify additional constraints using as an example the following MP problem:

$$\begin{aligned} &\text{Minimize } x + 2 * y \\ &\text{with: } 0 \leq x \leq 3 \\ &\qquad 0 \leq y \leq 3 \\ &\qquad x + y >= 1 \end{aligned}$$

The variables

The problem involves two variables, defined using the following clauses:

```
x[s]=v -> p(s),int[32](v), v<=3, v>=0.
p(s) -> x[s]=_ .

y[s]=v -> p(s),int[32](v), v<=3, v>=0.
p(s) -> y[s]=_ .

lang:solver:variable(`x).
lang:solver:variable(`y).

p(s),p:id(s:n) -> string(n).
```

Note that we have two lang:solver:variable declarations, one for each predicate we are trying to solve.

The objective function

The objective function is now derived from both *x* and *y*, and the goal of the computation will be to minimize its value:

```
objective[]=z -> int[32](z).
objective[]+=x[s]+2*y[s].
lang:solver:minimal(`objective).
```

The constraint

To ensure that the values of the variables always satisfy the constraint $x + y >= 1$, we add this constraint to the program using the clause:

```
x[s]=v1,y[s]=v2 -> v1+v2>=1.
```

The expected values

The program defines the expected value predicates for both variables:

```
x_expected[s]=v -> p(s),int[32](v).
y_expected[s]=v -> p(s),int[32](v).
```

Together, the above clauses for the *suite.program* block.

Result

To compute the solution, we add the *suite.program* block to a workspace, and populate the EDB by executing *test.init*:

```
+p(s),+p:id(s:"s").

+x_expected[s]=1 <- ^p(s).
+y_expected[s]=0 <- ^p(s).
```

Consequently, the predicates *x[]* and *y[]* are populated with the values which solve our problem. The result is tested by the *suite.assert* block:

```
+system:bloxunit:Compare:expected[c]="x_expected",
+system:bloxunit:Compare:actual[c]="x" <- system:bloxunit:Compare:id[c]=0.

+system:bloxunit:Compare:expected[c]="y_expected",
+system:bloxunit:Compare:actual[c]="y" <- system:bloxunit:Compare:id[c]=1.
```

2.3 Indexed Variables

In this section we reformulate the problem solved in the previous section so that the variables—rather than being different predicates—are different facts of one predicate. Consider the following problem statement:

$$\text{Minimize } \sum_i x_i * a_i$$
$$\text{with: } \forall i, 0 \leq x_i \leq 3$$
$$\sum_i x_i >= 1$$

The variables

The variables are now defined using one predicate, parameterized by an entity *index* with an integer refmod:

```
x[i]=v -> index(i), int[32](v), v<=3, v>=0.
index(i) -> x[i]=_.
lang:solver:variable(`x).
index(i),index:id[i]=n -> uint[32](n), lang:inv[index:id][n]=i.
```

Above, the first and second clauses declare and define the variable predicate, and the third clause declares the *index* entity, which serves as the type of the variable predicate's key argument. Note that the key argument type is the only thing that differs this variable predicate specification from the specifications given in the previous sections.

The parameters

The numbers 2 and 1, which define the objective function $x + 2 * y$ in the previous section, are represented by an EDB predicate *a* keyed—just as the variable predicate—by the *index* entity:

```
a[i]=n -> index(i),int[32](n).
```

The objective function

The objective function is now an aggregation of the multiplication of the corresponding values of the variable and parameter predicates:

```
lang:solver:minimal(`objective).
objective[]=z -> int[32](z).
objective[]+=x[i]*a[i].
```

The constraint

The constraint is defined using an auxiliary predicate aggregating the values of the variable predicate:

```
sum[]=n -> int[32](n).
sum[]+=x[_].
sum[]=v -> v>=1.
```

The expected value

The expected value predicate also ranges over the *index* entity:

```
x_expected[i]=v -> index(i),int[32](v).
```

Together, the above clauses form the *suite.program* block.

The computation

When the *suite.program* block is added to a workspace, we run the program by executing the *test.init* block of the form:

```
+index(i),+index:id[i]=0.
+index(i),+index:id[i]=1.

+a[i]=1 <- +index:id[i]=0.
+a[i]=2 <- +index:id[i]=1.

+x_expected[i]=1 <- +index:id[i]=0.
+x_expected[i]=0 <- +index:id[i]=1.
```

Consequently, the predicate *x* is populated, for all values of *index*, with values which solve the problem. We can test the result with the *test.assert* block:

```
+system:bloxunit:Compare:expected[c] = "x_expected",
+system:bloxunit:Compare:actual[c] = "x" <- system:bloxunit:Compare:id(c:0).
```

The re-computation

The abstract specification presented in this section enables solving the same problem for different parameter sets without changing the program logic. We can do so by following the execution of *test.init* with the execution of the *test2.init* block:

```
^a[i]=2 <- index:id@previous[i]=0.
^a[i]=1 <- index:id@previous[i]=1.

^x_expected[i]=0 <- index:id@previous[i]=0.
^x_expected[i]=1 <- index:id@previous[i]=1.

+index(i),+index:id[i]=2.
+a[i]=3 <- +index:id[i]=2.
+x_expected[i]=0 <- +index:id[i]=2.
```

Above, the first two clauses switch the values of the parameters set in the first test, whereas the clauses third and fourth accordingly update the expected values of the variables. The last three clauses extend the problem with another variable by adding new instance of the *index* entity, and setting the parameter and the expected value for that index.

2.4 Indexed Constraints

Just like the variables can be indexed by a Datalog predicate, the constraints can be generated by an index. In this section we illustrate this feature by extending the problem from the previous section with a set of variables linked by constraints to the variables from the previous problem:

$$\begin{aligned} &\text{Minimize } \sum_i x_i * a_i \\ &\text{with: } \forall i, 0 \leq x_i \leq 3 \\ &\qquad \forall i, 0 \leq y_i \leq 3 \\ &\qquad \sum_i x_i >= 1 \\ &\qquad \forall i, y_i = x_i + 1 \end{aligned}$$

The variables

We define two sets of variables keyed by the same index set *index*:

```
index(i),index:id[i]=n -> uint[32](n), lang:inv[index:id][n]=i.

x[i]=v -> index(i), int[32](v), v<=3, v>=0.
index(i) -> x[i]=_.

y[i]=v -> index(i), int[32](v), v<=3, v>=0.
index(i) -> y[i]=_ .
lang:solver:variable(`x).
lang:solver:variable(`y).
```

The parameters, expected values and objective function

The specification of the parameters, the expected value predicates and the objective function remains the same as before:

```
param[i]=n -> index(i), int[32](n).

x_expected[i]=v -> index(i), int[32](v).
y_expected[i]=v -> index(i), int[32](v).

lang:solver:minimal(`objective).
objective[]=z -> int[32](z).
objective[]+=x[i]*param[i].
```

The constraints

The first constraint is carried from the previous example:

```
sum[]=n -> int[32](n).
sum[]+=x[_].
sum[]=v -> v>=1.
```

The second constraint applies to both variable predicates for each instance of the *index* entity:

```
// original version, working
index(i),x[i]-y[i]=v -> v=-1.
```

Note the encoding of the constraint compared to its mathematical form at the beginning of the section. Current limitations of BloxOptimize require that variable predicates appear only in constraint bodies.

Together, the above clauses form the *suite.program* block.

The computation

When the *suite.program* block is added to a workspace, we solve the example by running the *test.init* block:

```
+index(i),+index:id[i]=0.
+index(i),+index:id[i]=1.

+param[i]=1<-+index:id[i]=0.
+param[i]=2<-+index:id[i]=1.

+x_expected[i]=1<-+index:id[i]=0.
+x_expected[i]=0<-+index:id[i]=1.

+y_expected[i]=2<-+index:id[i]=0.
+y_expected[i]=1<-+index:id[i]=1.
```

Consequently, for all values of *index*, the predicates *x* and *y* are populated with the values which solve the problem.

The result is tested by the *test.assert* block:

```
+system:bloxunit:Compare:expected[c]= "x_expected",
+system:bloxunit:Compare:actual[c]="x"<- system:bloxunit:Compare:id[c]=0.

+system:bloxunit:Compare:expected[c]= "y_expected",
+system:bloxunit:Compare:actual[c]="y"<- system:bloxunit:Compare:id[c]=1.
```

2.5 Filtered Constraints

Indexed constraints may apply only to a subset of the type by which they are indexed, as illustrated in this section.

The variables

We define a set of variables:

```
index(i),index:id(i:id) -> uint[32](id).

x[i]=v -> index(i), v>=0, int[32](v).
index(i) -> x[i]=_.
lang:solver:variable(`x).
```

and two filter predicates on the type indexing that variable set:

```
filter1(i) -> index(i).
filter1(i) <- index:id[i]<2.

filter2(i) -> index(i).
filter2(i) <- index:id[i]+1>=3.
```

The constraints

The above variable specification allows to define two sets of constraints, each applying only to a subset of the indices:

```
x[i]=v,filter1(i) -> v<=2.
x[i]=v, filter2(i)  -> v<=4.
```

The expected value and objective function predicates

The expected value and objective function predicates are defined as usual, completing the *suite.program* block:

```
x_expected[i]=v -> index(i),int[32](v).

objective[]=z -> int[32](z).
objective[]+=x[i].
lang:solver:maximal(`objective).
```

2.6 Disjunctive Constraints

Existing automated MP solvers lack direct support for problems involving disjunction. In order to be accepted by a solver, such problems must be first converted into non-disjunctive specifications using a dedicated algebraic transformation method. To accommodate problems with disjunctive constraints in LogicBlox, we added necessary extensions to the language syntax, and implemented two transformations for eliminating disjunction: convex hull (see *Convex Hull Transformation*) and big-M (see *Big-M Transformation*). We now present these extensions, as our first example using a slightly modified version of *The Simplest Problem*. The following sections discuss some of the more advanced syntactic features available for disjunctive problem specification in DatalogLB. Consider the problem:

$$\begin{aligned} &\text{Maximize } x \\ \text{with: } &0 \leq x \leq 3 \\ &(x \leq 2) \vee (x \geq 4) \end{aligned}$$

The variable

We use entity type *p* to declare the variable predicate *x* using the existential clause:

```
p(s),p:id(s:n) -> string(n).

x[s]=v -> p(s),int[32](v), v>=0, v<=4.
p(s) -> x[s]=_.
lang:solver:variable(`x).
```

The objective function and expected value predicates

We define the objective function and declare the expected value predicate in a usual manner:

```
lang:solver:maximal(`objective).
objective[]=z -> int[32](z).
objective[]+=x[_].

x_expected[s]=y -> p(s),int[32](y).
```

The constraint

We define the disjunctive constraint using the auxiliary predicate *constr*:

```
constr(s) -> p(s).
constr(s) <- x[s]>=1, x[s]<=2.
//constr(s) <- x[s]=3.
constr(s) <- x[s]>=4, x[s]<=5.
p(s) -> constr(s).
lang:isEntity[`constr]=false.
```

Above note the circular constraint specification formed by the first and fourth clauses. The *lang:isEntity* directive in the last clause prevents evaluation issues.

The above clauses form the *suite.program* block.

The computation

When *suite.program* is added to the workspace, the LogicBlox compiler detects the disjunctive constraint, and applies the convex hull transformation steps to eliminate the disjunction. The resulting program is then processed as a regular linear MIP problem. We execute the *test.init* block:

```
+p(s),+p:id(s:"s").
+x_expected[s] = 4 <- ^p(s).
```

The solver correctly computes variable value *x=2*, identical to the expected value in *test.init*.

2.7 Disjunctive Constraints—Conjunction within Disjuncts

Disjunctive constraints may contain conjunction inside the disjuncts. For instance, the following problem may be easily represented in DatalogLB:

$$\begin{aligned} & \text{Minimize } y \\ \text{with: } & 0 \leq x \leq 1 \\ & 0 \leq y \leq 10 \\ & (x = 0 \wedge y \leq 3) \vee (x = 1 \wedge y \geq 4) \end{aligned}$$

The variables, objective function and expected value predicates

We declare two variables using the existential clauses:

```
p(s),p:id(s:n) -> string(n).

x[s]=v -> p(s),int[32](v), v>=0, v<=1.
p(s) -> x[s]=_ .
lang:solver:variable(`x).

y[s]=v -> p(s),int[32](v), v>=0, v<=10.
p(s) -> y[s]=_ .
lang:solver:variable(`y).
```

We define the objective function to minimize the variable *y* and define the expected value predicates:

```
objective[]=z -> int[32](z).
objective[]+=y[_].
lang:solver:maximal(`objective).

x_expected[s]=v -> p(s),int[32](v).
y_expected[s]=v -> p(s),int[32](v).
```

The constraint

We use a right-arrow rule to constrain the variables with $(x = 0 \wedge y \leq 3) \vee (x = 1 \wedge y \geq 4)$:

```
p(s) -> constr(s).
constr(s) <- x[s]<=0, y[s]<=3.
constr(s) <- x[s]>=1, y[s]>=4.
constr(s) -> p(s).
lang:isEntity[`constr]=false.
```

Note the encoding of $x = 0$ and $x = 1$. Current limitations of BloxOptimize require representing equality expressions in terms of the $\leq$ and $\geq$ relations.

The above clauses form the *suite.program* block.

The computation

When *suite.program* is added to the workspace, we solve the problem by executing the *test.init* block:

```
+p(s),+p:id(s:"s").

+x_expected[s]=1 <- ^p(s).
+y_expected[s]=10 <- ^p(s).
```

The solver correctly computes variable values *x=1* and *y=10*, identical to the expected values in *test.init*.

2.8 Disjunctive Constraints—Filtered Domain Types

As with the standard linear programs, disjunctive programs admit indexed constraints that apply only to subsets of types by which they are indexed, as we demonstrate in this section.

The variable

We use entity type *index* to declare the indexed variable predicate *x* using the existential clause:

```
index(i),index:id(i:id) -> uint[32](id).

x[i]=v -> index(i), v>=0, v<=10, int[32](v).
index(i) -> x[i]=_.
lang:solver:variable(`x).
```

In the program we consider two subsets of the *index* type, defined by two filter predicates:

```
filter1(i) -> index(i).
filter1(i) <- index:id[i]<2.

filter2(i) -> index(i).
filter2(i) <- index:id[i]+1>=3.
```

The objective function and expected value predicates

We define the objective function and declare the expected value predicate in a usual manner:

```
objective[]=z -> int[32](z).
objective[]+=x[_].
lang:solver:maximal(`objective).

x_expected[i]=v -> index(i),int[32](v).
```

The constraints

We define the disjunctive constraints using the auxiliary predicates *p1* and *p2* indexed by the two filter types:

```
p1(i) -> index(i).
p1(i) <- x[i]<=2.
p1(i) <- x[i]>=4.
filter1(i) -> p1(i).

p2(i) -> index(i).
p2(i) <- x[i]<=8.
p2(i) <- x[i]>=12.
filter2(i) -> p2(i).
```

The above clauses form the *suite.program* block.

The computation

When *suite.program* is added to the workspace, we solve the problem by executing the *test.init* block:

```
+index(i),+index:id[i]=0.
+index(i),+index:id[i]=1.
+index(i),+index:id[i]=2.
+index(i),+index:id[i]=3.

+x_expected[i]=10 <- +index:id[i]=0.
+x_expected[i]=10 <- +index:id[i]=1.
+x_expected[i]=8 <- +index:id[i]=2.
+x_expected[i]=8 <- +index:id[i]=3.
```

The solver correctly computes variable values, identical to the expected values in *test.init*.

2.9 Disjunctive Constraints—Multiple Variables

Disjunctive MP programs may involve more than one variable. For example, consider the following problem:

$$\begin{aligned} & \text{Maximize } x + 2y \\ \text{with: } & 0 \leq x \leq 3 \\ & 0 \leq y \leq 3 \\ & (x + y \geq 7) \vee (x + y \leq 4) \end{aligned}$$

The variables

We use entity type *p* to declare two variables using the existential clauses:

```
p(s),p:id(s:n) -> string(n).

x[s]=v -> p(s),int[32](v), v>=0, v<=3.
p(s) -> x[s]=_ .
lang:solver:variable(`x).

y[s]=v -> p(s),int[32](v), v>=0, v<=3.
p(s) -> y[s]=_ .
lang:solver:variable(`y).
```

The objective function and expected value predicates

We define the objective function to compute the value of the expression *x+2y* and declare the expected value predicates:

```
objective[]=z -> int[32](z).
objective[]+=x[s]+2*y[s].
lang:solver:maximal(`objective).

x_expected[s]=v -> p(s),int[32](v).
y_expected[s]=v -> p(s),int[32](v).
```

The constraints

We define the disjunctive constraint using the auxiliary predicate *constr*:

```
p(s) -> constr(s).
//constr(s) <- x[s]+y[s]>=7.
constr(s) <- x[s]+y[s]<=4.
constr(s) -> p(s).
lang:isEntity[`constr]=false.
```

We further constrain the values of the variables to satisfy $x \leq y$ with the rule:

```
x[s]=v1,y[s]=v2 -> v1<=v2.
```

The above clauses form the *suite.program* block.

The computation

When *suite.program* is added to the workspace, we solve the problem by executing the *test.init* block:

```
+p(s),+p:id(s:"s").

+x_expected[s]=1 <- ^p(s).
+y_expected[s]=3 <- ^p(s).
```

The solver correctly computes variable values, identical to the expected values in *test.init*.

2.10 Disjunctive Constraints—Nested Disjunction

The LogicBlox compiler accommodates disjunctive problems with more than two disjuncts per constraint, as shown by the following example. Not surprisingly, each disjunct contributes to a significant increase in the size of the transformed problem, thus resulting in slower evaluation. At this time our system can successfully solve problems with up to five disjuncts per constraint.

Consider the problem:

$$\text{Maximize } x$$
$$\text{with: } 0 \leq x \leq 8$$
$$(x \leq 1) \vee (x \geq 3 \wedge x \leq 6) \vee (x \geq 9)$$

The variable

We use entity type *p* to declare the variable predicate *x* using the existential clause:

```
p(s),p:id(s:n) -> string(n).

x[s]=v -> p(s),int[32](v), v>=0, v<=8.
p(s) -> x[s]=_.
lang:solver:variable(`x).
```

The objective function and expected value predicates

We define the objective function and declare the expected value predicate in the usual manner:

```
lang:solver:maximal(`objective).
objective[]=z -> int[32](z).
objective[]+=x[_].

x_expected[s]=y -> p(s),int[32](y).
```

The constraint

We define the disjunctive constraint using the auxiliary predicate *constr*:

```
constr(s) -> p(s).


// For convex hull,
// this works
constr(s) <- x[s]<=1.
constr(s) <- x[s]>=3,x[s]<=6.
constr(s) <- x[s]>=9.


/*
// and this raises BTreeException: data too large
constr(s) <- x[s]<=1.
constr(s) <- x[s]>=2,x[s]<=3.
constr(s) <- x[s]>=4,x[s]<=5.
constr(s) <- x[s]>=6,x[s]<=7.
constr(s) <- x[s]>=8,x[s]<=9.
```

```
constr(s) <- x[s]>=10.
*/

/*
// For big-M,
// this works ...
constr(s) <- x[s]<=1.
constr(s) <- x[s]>=2,x[s]<=3.
constr(s) <- x[s]>=4,x[s]<=5.
constr(s) <- x[s]>=6,x[s]<=7.
constr(s) <- x[s]>=8,x[s]<=9.
constr(s) <- x[s]>=10,x[s]<=11.
constr(s) <- x[s]>=12,x[s]<=13.
constr(s) <- x[s]>=14,x[s]<=15.
constr(s) <- x[s]>=16,x[s]<=17.
constr(s) <- x[s]>=18,x[s]<=19.
constr(s) <- x[s]>=20,x[s]<=21.
constr(s) <- x[s]>=22,x[s]<=23.
constr(s) <- x[s]>=24,x[s]<=25.
constr(s) <- x[s]>=26,x[s]<=27.
constr(s) <- x[s]>=28,x[s]<=29.
constr(s) <- x[s]>=30,x[s]<=31.
constr(s) <- x[s]>=32,x[s]<=33.
constr(s) <- x[s]>=34,x[s]<=35.
constr(s) <- x[s]>=36,x[s]<=37.
constr(s) <- x[s]>=38,x[s]<=39.
constr(s) <- x[s]>=40,x[s]<=41.
constr(s) <- x[s]>=42,x[s]<=43.
constr(s) <- x[s]>=44,x[s]<=45.
constr(s) <- x[s]>=46,x[s]<=47.
constr(s) <- x[s]>=48,x[s]<=49.
constr(s) <- x[s]>=50,x[s]<=51.
constr(s) <- x[s]>=52,x[s]<=53.
constr(s) <- x[s]>=54,x[s]<=55.
constr(s) <- x[s]>=56,x[s]<=57.
constr(s) <- x[s]>=58,x[s]<=59.
constr(s) <- x[s]>=60.
*/


p(s) -> constr(s).
lang:isEntity[`constr]=false.
optimization:disjunctionRewrite[]="big M".
```

The above clauses form the *suite.program* block.

The computation

When *suite.program* is added to the workspace, the LogicBlox compiler detects the disjunctive constraint, and applies the convex hull transformation steps to eliminate the disjunction. The resulting program is then processed as a regular linear problem. We execute the *test.init* block:

```
+p(s),+p:id(s:"s").
+x_expected[s] = 6 <- ^p(s).
```

The solver correctly computes the value for the variable *x*, identical to the expected value in *test.init*.

2.11 Dual Values

In this section we further extend our implementation to allow providing dual values for the constraints. As dual value computation is the standard functionality of the MP solvers, our main task to facilitate this extension is to declare appropriate predicates to read and store dual values obtained by the solvers. Such a dual predicate should be a function and should have a set of indexes corresponding to the structure of the universal quantifiers of the constraint for which the dual is computed. The domain of the dual predicate should be of type *float[64]*.

Dual values can be stored in a predetermined predicate if their corresponding constraints are expressed using temporary predicates defined with a dedicated predicate-to-predicate (p2p) mapping *dual '. The p2p 'dual* has no logical meaning, except for tagging the constraints and obtaining their dual values. Thus, given a constraint *FOO(i), X[i]=v -> v=0*, we can compute its dual value in the following sequence of steps:

- create a temporary predicate *t(i)* indexed by the universally quantified variables of the constraint;
- define the predicate as a rule using the p2p *dual* of the form:

  t(i) <- dual<<dualPredName,i>> body

where *body* is the constraint *X[i]>=0*;

- substitute *q(i)* as the head of the original constraint:

  FOO(i) -> t(i).

For example, the following is the encoding of the dual predicate in our test suite *suiteDual*:

```
dualNutr(n;v) -> NUTR(n), float[64](v).
totalNutrAmt[n]=v -> NUTR(n), float[64](v).
totalNutrAmt[n]+=Buy[f]*amt[n,f].
NUTR(n) -> q(n).
q(n) -> NUTR(n).
q(n) <- dual<<dualNutr,n>> totalNutrAmt[n]>=nutrLow[n].
```

This is the same constraint as:

```
totalNutrAmt[n]=v -> NUTR(n), float[64](v).
totalNutrAmt[n]+=Buy[f]*amt[n,f].
NUTR(n),totalNutrAmt[n]=v1,nutrLow[n]=v2 -> v1>=v2.
```

Note that:

- In order to specify where to store the value of the dual variables, the constraint needs to be defined as a temporary predicate *q(n)*
- The constraint *NUTR(n)->q(n)* would be detected as a circular type dependency. To avoid this, we need to enhance our program with the *lang:isEntity* directive to warn the compiler that *q* is not an entity:

  ```
  lang:isEntity[`q]=false.
  ```

2.12 Indexed Objective Function

Another extension of our framework allows to decompose problems of certain form into simpler sub-problems, which can be handled by the solver independently of each other. Consider the following problem:

$$\text{Maximize } \forall_{i \in p} \sum_{j \in r} x_{ij} * a_j$$
$$\text{with: } \forall_{i \in p} \forall_{j \in r}, 0 \leq x_{ij} \leq 3$$
$$\forall_{i \in p} \forall_{j \in r}, x_{ij} \leq b_i$$

The problem is encoded in our language as follows:

CHAPTER

THREE

APPLICATIONS

3.1 Diet Problem (suiteDiet)

Scenario: Stella works as a nutrition specialist in a fast food restaurant chain TastyMac. One of her tasks, aimed at improving efficiency of ordering the food, is to compose individual food items into meal packages that are both inexpensive and nutritious. Given a set of food products, their prices and nutritional contents, as well as the minimum required amounts for a set of nutrients, Stella wants to plan meals that meet the nutritional requirements at the possibly lowest cost. Thus, Stella writes the following problem specification.

Problem Specification:

Each *parameter* represents some externally provided information:

- *FOOD*—set of food products
- *NUTR*—set of nutrients
- *amt(n,f)*— amount of a nutrient *n* in food product *f*
- *nutrLow(n)*—minimum required amount of a nutrient *n*
- *cost(f)*—cost of food product *f*

The *variable* represents the value to be computed:

- *Buy(f)*—the amount of food *f* to be purchased for the meal

The *objective function* defines the total cost of all purchased food products:

- $\sum_{f \in FOOD} Buy(f) * cost(f)$

For each nutrient, the *constraint* requires that the amount of the nutrient in the purchased meal is not lower than the minimum required amount of that nutrient:

- $\forall n \in NUTR, \sum_{f \in FOOD} Buy(f) * amt(n, f) \geq nutrLow(n)$

Installed Program

The above problem specification is represented as a program in DatalogLB, in which the parameters are defined as:

```
//sets
NUTR(x), NUTR:name(x:n) -> string(n).
FOOD(x), FOOD:name(x:n) -> string(n).

//parameters
amt[n,f]=a -> NUTR(n), FOOD(f), float[64](a), a>=0.
nutrLow[n]=b -> NUTR(n), float[64](b), b>=0.
cost[f]=c -> FOOD(f), float[64](c), c>=0.
```

The variable is indexed by food products:

```
Buy[f]=b -> FOOD(f), float[64](b), b>=0.
FOOD(f) -> Buy[f]=_.
lang:solver:variable(`Buy).
```

The goal of the computation—to minimize the objective function $\sum_{f \in FOOD} Buy(f) * cost(f)$—is encoded as:

```
lang:solver:minimal(`objective).
objective[]=v -> float[64](v).
objective[]+=Buy[f]*cost[f].
```

The constraint $\forall n \in NUTR, \sum_{f \in FOOD} Buy(f) * amt(n, f) \geq nutrLow(n)$ is encoded as:

```
totalNutrAmt[n]=v -> NUTR(n), float[64](v).
totalNutrAmt[n]+=Buy[f]*amt[n,f].
NUTR(n),totalNutrAmt[n]=v1,nutrLow[n]=v2 -> v1>=v2.
```

The program block *suite.program* is completed by the definition of the expected value predicates:

```
Buy_expected[f]=b -> FOOD(f), float[64](b), b>=0.
```

The computation:

When the *suite.program* block is loaded to a workspace, we compute the result by executing the *test.init* block which provides the program with the values of the index sets and their associated parameters:

```
+NUTR(x), +NUTR:name(x:"Prot").
+NUTR(x), +NUTR:name(x:"Iron").
+NUTR(x), +NUTR:name(x:"VitA").
+NUTR(x), +NUTR:name(x:"Cals").
+NUTR(x), +NUTR:name(x:"VitC").
+NUTR(x), +NUTR:name(x:"Carb").
+NUTR(x), +NUTR:name(x:"Calc").

+FOOD(x), +FOOD:name(x:"QP").
+FOOD(x), +FOOD:name(x:"MD").
+FOOD(x), +FOOD:name(x:"FR").
+FOOD(x), +FOOD:name(x:"SM").
+FOOD(x), +FOOD:name(x:"BM").
+FOOD(x), +FOOD:name(x:"1M").
+FOOD(x), +FOOD:name(x:"FF").
+FOOD(x), +FOOD:name(x:"OJ").
+FOOD(x), +FOOD:name(x:"MC").

+nutrLow[x] = 55 <- NUTR:name(x:"Prot").
+nutrLow[x] = 100 <- NUTR:name(x:"VitA").
+nutrLow[x] = 100 <- NUTR:name(x:"VitC").
+nutrLow[x] = 100 <- NUTR:name(x:"Calc").
+nutrLow[x] = 100 <- NUTR:name(x:"Iron").
+nutrLow[x] = 2000 <- NUTR:name(x:"Cals").
+nutrLow[x] = 350 <- NUTR:name(x:"Carb").

+cost[f] = 1.84 <- FOOD:name(f:"QP").
+cost[f] = 2.19 <- FOOD:name(f:"MD").
+cost[f] = 1.84 <- FOOD:name(f:"BM").
+cost[f] = 1.44 <- FOOD:name(f:"FF").
+cost[f] = 2.29 <- FOOD:name(f:"MC").
```

```
+cost[f] = 0.77 <- FOOD:name(f:"FR").
+cost[f] = 1.29 <- FOOD:name(f:"SM").
+cost[f] = 0.60 <- FOOD:name(f:"1M").
+cost[f] = 0.72 <- FOOD:name(f:"OJ").

+amt[n,x] = a <- NUTR:name(n:"Cals"), FOOD:name(x:"QP"), a=510.
+amt[n,x] = a <- NUTR:name(n:"Cals"), FOOD:name(x:"MD"), a=370.
+amt[n,x] = a <- NUTR:name(n:"Cals"), FOOD:name(x:"BM"), a=500.
+amt[n,x] = a <- NUTR:name(n:"Cals"), FOOD:name(x:"FF"), a=370.
+amt[n,x] = a <- NUTR:name(n:"Cals"), FOOD:name(x:"MC"), a=400.
+amt[n,x] = a <- NUTR:name(n:"Cals"), FOOD:name(x:"FR"), a=220.
+amt[n,x] = a <- NUTR:name(n:"Cals"), FOOD:name(x:"SM"), a=345.
+amt[n,x] = a <- NUTR:name(n:"Cals"), FOOD:name(x:"1M"), a=110.
+amt[n,x] = a <- NUTR:name(n:"Cals"), FOOD:name(x:"OJ"), a=80.

+amt[n,x] = a <- NUTR:name(n:"Carb"), FOOD:name(x:"QP"), a=34.
+amt[n,x] = a <- NUTR:name(n:"Carb"), FOOD:name(x:"MD"), a=35.
+amt[n,x] = a <- NUTR:name(n:"Carb"), FOOD:name(x:"BM"), a=42.
+amt[n,x] = a <- NUTR:name(n:"Carb"), FOOD:name(x:"FF"), a=38.
+amt[n,x] = a <- NUTR:name(n:"Carb"), FOOD:name(x:"MC"), a=42.
+amt[n,x] = a <- NUTR:name(n:"Carb"), FOOD:name(x:"FR"), a=26.
+amt[n,x] = a <- NUTR:name(n:"Carb"), FOOD:name(x:"SM"), a=27.
+amt[n,x] = a <- NUTR:name(n:"Carb"), FOOD:name(x:"1M"), a=12.
+amt[n,x] = a <- NUTR:name(n:"Carb"), FOOD:name(x:"OJ"), a=20.

+amt[n,x] = a <- NUTR:name(n:"Prot"), FOOD:name(x:"QP"), a=28.
+amt[n,x] = a <- NUTR:name(n:"Prot"), FOOD:name(x:"MD"), a=24.
+amt[n,x] = a <- NUTR:name(n:"Prot"), FOOD:name(x:"BM"), a=25.
+amt[n,x] = a <- NUTR:name(n:"Prot"), FOOD:name(x:"FF"), a=14.
+amt[n,x] = a <- NUTR:name(n:"Prot"), FOOD:name(x:"MC"), a=31.
+amt[n,x] = a <- NUTR:name(n:"Prot"), FOOD:name(x:"FR"), a=3.
+amt[n,x] = a <- NUTR:name(n:"Prot"), FOOD:name(x:"SM"), a=15.
+amt[n,x] = a <- NUTR:name(n:"Prot"), FOOD:name(x:"1M"), a=9.
+amt[n,x] = a <- NUTR:name(n:"Prot"), FOOD:name(x:"OJ"), a=1.

+amt[n,x] = a <- NUTR:name(n:"VitA"), FOOD:name(x:"QP"), a=15.
+amt[n,x] = a <- NUTR:name(n:"VitA"), FOOD:name(x:"MD"), a=15.
+amt[n,x] = a <- NUTR:name(n:"VitA"), FOOD:name(x:"BM"), a=6.
+amt[n,x] = a <- NUTR:name(n:"VitA"), FOOD:name(x:"FF"), a=2.
+amt[n,x] = a <- NUTR:name(n:"VitA"), FOOD:name(x:"MC"), a=8.
+amt[n,x] = a <- NUTR:name(n:"VitA"), FOOD:name(x:"FR"), a=0.
+amt[n,x] = a <- NUTR:name(n:"VitA"), FOOD:name(x:"SM"), a=4.
+amt[n,x] = a <- NUTR:name(n:"VitA"), FOOD:name(x:"1M"), a=10.
+amt[n,x] = a <- NUTR:name(n:"VitA"), FOOD:name(x:"OJ"), a=2.

+amt[n,x] = a <- NUTR:name(n:"VitC"), FOOD:name(x:"QP"), a=6.
+amt[n,x] = a <- NUTR:name(n:"VitC"), FOOD:name(x:"MD"), a=10.
+amt[n,x] = a <- NUTR:name(n:"VitC"), FOOD:name(x:"BM"), a=2.
+amt[n,x] = a <- NUTR:name(n:"VitC"), FOOD:name(x:"FF"), a=0.
+amt[n,x] = a <- NUTR:name(n:"VitC"), FOOD:name(x:"MC"), a=15.
+amt[n,x] = a <- NUTR:name(n:"VitC"), FOOD:name(x:"FR"), a=15.
+amt[n,x] = a <- NUTR:name(n:"VitC"), FOOD:name(x:"SM"), a=0.
+amt[n,x] = a <- NUTR:name(n:"VitC"), FOOD:name(x:"1M"), a=4.
+amt[n,x] = a <- NUTR:name(n:"VitC"), FOOD:name(x:"OJ"), a=120.

+amt[n,x] = a <- NUTR:name(n:"Calc"), FOOD:name(x:"QP"), a=30.
+amt[n,x] = a <- NUTR:name(n:"Calc"), FOOD:name(x:"MD"), a=20.
+amt[n,x] = a <- NUTR:name(n:"Calc"), FOOD:name(x:"BM"), a=25.
```

```
+amt[n,x] = a <- NUTR:name(n:"Calc"), FOOD:name(x:"FF"), a=15.
+amt[n,x] = a <- NUTR:name(n:"Calc"), FOOD:name(x:"MC"), a=15.
+amt[n,x] = a <- NUTR:name(n:"Calc"), FOOD:name(x:"FR"), a=0.
+amt[n,x] = a <- NUTR:name(n:"Calc"), FOOD:name(x:"SM"), a=20.
+amt[n,x] = a <- NUTR:name(n:"Calc"), FOOD:name(x:"1M"), a=30.
+amt[n,x] = a <- NUTR:name(n:"Calc"), FOOD:name(x:"OJ"), a=2.

+amt[n,x] = a <- NUTR:name(n:"Iron"), FOOD:name(x:"QP"), a=20.
+amt[n,x] = a <- NUTR:name(n:"Iron"), FOOD:name(x:"MD"), a=20.
+amt[n,x] = a <- NUTR:name(n:"Iron"), FOOD:name(x:"BM"), a=20.
+amt[n,x] = a <- NUTR:name(n:"Iron"), FOOD:name(x:"FF"), a=10.
+amt[n,x] = a <- NUTR:name(n:"Iron"), FOOD:name(x:"MC"), a=8.
+amt[n,x] = a <- NUTR:name(n:"Iron"), FOOD:name(x:"FR"), a=2.
+amt[n,x] = a <- NUTR:name(n:"Iron"), FOOD:name(x:"SM"), a=15.
+amt[n,x] = a <- NUTR:name(n:"Iron"), FOOD:name(x:"1M"), a=0.
+amt[n,x] = a <- NUTR:name(n:"Iron"), FOOD:name(x:"OJ"), a=2.

+Buy_expected[f] = 4.38525 <- FOOD:name(f:"QP").
+Buy_expected[f] = 0 <- FOOD:name(f:"MD").
+Buy_expected[f] = 6.14754 <- FOOD:name(f:"FR").
+Buy_expected[f] = 0 <- FOOD:name(f:"FF").
+Buy_expected[f] = 0 <- FOOD:name(f:"SM").
+Buy_expected[f] = 0 <- FOOD:name(f:"BM").
+Buy_expected[f] = 3.42213 <- FOOD:name(f:"1M").
+Buy_expected[f] = 0 <- FOOD:name(f:"OJ").
+Buy_expected[f] = 0 <- FOOD:name(f:"MC").
```

The solver computes the values for the variable `Buy[f]`. The result is tested by the *test.assert* block:

```
+system:bloxunit:Compare:expected[c] = "Buy_expected",
+system:bloxunit:Compare:actual[c] = "Buy" <- system:bloxunit:Compare:id(c:0).
```

3.2 Diet Problem as a set of subproblems (suiteCustomDiet)

Scenario: Encouraged by the success of TastyMac meal packages, the company executives decide to extend the offer to accommodate various dietary needs of their customers. Stella's task is now to provide various combinations of the TastyMac menu items to satisfy nutritional requirements for different lifestyles. Thus, Stella adds to her original program another dimension representing such custom requirements.

The index sets

The new encoding involves the additional entity *DIET* representing nutritional needs for different lifestyles:

```
**The variables**
```

Compared to the original specification, the variable predicate has now an extra dimension:

```
Buy[f,d]=v -> FOOD(f), DIET(d), float[64](v), v>=0.
FOOD(f), DIET(d) -> Buy[f,d]=_.
lang:solver:variable(`Buy).
```

The objective function

The objective function is no longer a scalar, but a function of the variable which serves as the index to the problem:

```
lang:solver:minimal(`objective).
objective[d]=v -> DIET(d), float[64](v).
objective[d]+=Buy[f,d]*cost[f].
```

The constraints

The constraints also accommodate the new dimension:

```
totalNutrAmt[n,d]=v -> NUTR(n), DIET(d), float[64](v).
totalNutrAmt[n,d]+=Buy[f,d]*amt[n,f].
totalNutrAmt[n,d]=v1, nutrLow[n,d]=v2 -> v1>=v2.
```

Some constraints may involve predicates which filter along the new dimension. Such constraints apply only to some instances of the problem:

```
hasMeat[d]=v -> DIET(d), float[64](v).
hasMeat[d]+=Buy[f,d]*cost[f].
highProtein(d) <- DIET:name(d:"athlete").
highProtein(d), hasMeat[d]=v -> v>=30.
```

Our framework decomposes the resulting program into several sub-problems, one for each set of dietary constraints, as described in Section *Indexed Objective Function*.

3.3 Production Problem (suiteProduction)

Scenario: Ben runs a factory manufacturing a variety of products. Each product requires certain factory time to be manufactured, and provides certain profit. Ben wants to maximize the total profit that the factory can generate during given amount of available factory time, and make sure that the manufactured amount of each product does not exceed certain limit.

Problem Specification:

The entity *PRODUCT* denotes the set of products manufactured at the factory.

Each *parameter* stores some externally provided information:

- *tonsPerHr(p)*—number of tons of product *p* that can be manufactured in one hour
- *profitPerTon(p)*—profit provided by one ton of product *p*
- *maxTons(p)*—the upper production limit for product *p*
- *maxHrs*—number of available factory hours

The *variable Tons(p)* represents the number of tons of product *p* to be manufactured.

The *objective function* defines the total profit generated at the factory:

- $\sum_{p \in PROD} Tons(p) * profitPerTon(p)$

Each *constraint* places a restriction on the manufactured product amount or the production time

- for each product, the manufactured amount of the product is non-negative, and does not exceed that product's production limit:
  - $\forall p \in PROD, 0 \leq Tons(p) \leq maxTons(p)$
- the total production time does not exceed the available factory time:
  - $\sum_{p \in PROD} \frac{1}{tonsPerHr(p)} \leq maxHrs$

Installed Program

The entity and the parameters of the problem are defined as:

```
//sets
PRODUCT(x), PRODUCT:name(x:n) -> string(n).

//parameters
tonsPerHr[p]=v -> PRODUCT(p), float[64](v).
hrsPerTon[p]=v -> PRODUCT(p), float[64](v).
hrsPerTon[p]=1/tonsPerHr[p].
profitPerTon[p]=v -> PRODUCT(p), float[64](v).
maxTons[p]=v -> PRODUCT(p), float[64](v).
maxHrs[]=v -> float[64](v).
```

Note that the parameter *hrsPerTon*, representing the production time of a product, is calculated from the EDB predicate *tonsPerHr*.

The variable is indexed by the products, and its declaration includes the constraint $\forall p \in PROD, Tons(p) \geq 0$:

```
Tons[p]=v -> PRODUCT(p), float[64](v), v>=0.
PRODUCT(p) -> Tons[p]=_ .
lang:solver:variable('Tons).
```

The goal of the computation—to maximize the objective function $\sum_{p \in PROD} Tons(p) * profitPerTon(p)$—is encoded as:

```
//objective
lang:solver:maximal('TotalProfit).
TotalProfit[]=v -> float[64](v).
TotalProfit[]+=Tons[p]*profitPerTon[p].
```

The constraint $\forall p \in PROD, Tons(p) \leq maxTons(p)$ is encoded as:

```
PRODUCT(p) -> Tons[p]<=maxTons[p].
```

The constraint $\sum_{p \in PROD} \frac{1}{tonsPerHr(p)} \leq maxHrs$ is encoded as:

```
totalProductHrs[]=v -> float[64](v).
totalProductHrs[]+=Tons[p]*hrsPerTon[p].
totalProductHrs[]=v1, maxHrs[]=v2 -> v1<=v2.
```

Execution

The program is executed for given index sets along with their associated parameters, for example:

```
+PRODUCT(x), +PRODUCT:name(x:"bands").
+PRODUCT(x), +PRODUCT:name(x:"coils").

+tonsPerHr[j] = 200 <- PRODUCT:name(j:"bands").
+tonsPerHr[j] = 140 <- PRODUCT:name(j:"coils").

+profitPerTon[j] = 25 <- PRODUCT:name(j:"bands").
+profitPerTon[j] = 30 <- PRODUCT:name(j:"coils").

+maxTons[j] = 6000 <- PRODUCT:name(j:"bands").
+maxTons[j] = 4000 <- PRODUCT:name(j:"coils").
```

```
+maxHrs[]=40.

+Tons_expected[j] = 6000 <- PRODUCT:name(j:"bands").
+Tons_expected[j] = 1400 <- PRODUCT:name(j:"coils").
```

The solver computes the values for the variable `Tons[p]`. The result is tested by:

```
+system:bloxunit:Compare:expected[c] = "Tons_expected",
+system:bloxunit:Compare:actual[c] = "Tons" <- system:bloxunit:Compare:id(c:0).
```

3.4 Production Problem (version 2, suiteProductionMaxMin)

Scenario: Ben wants his factory to respond to product demand and maintain at least a certain minimal production level for each product. Thus, he extends the previous model with a new parameter—a minimum amount of each product that has to be manufactured.

Problem Specification:

The entity *PRODUCT* denotes the set of products manufactured at the factory.

Each *parameter* stores some externally provided information:

- *tonsPerHr(p)*—number of tons of product *p* that can be manufactured in one hour
- *profitPerTon(p)*—profit provided by one ton of product *p*
- *minTons(p)*—the lower production limit for product *p*
- *maxTons(p)*—the upper production limit for product *p*
- *maxHrs*—number of available factory hours

The *variable Tons(p)* represents the number of tons of product *p* to be manufactured.

The *objective function* defines the total profit generated at the factory:

- $\sum_{p \in PRODUCT} Tons(p) * profitPerTon(p)$

Each *constraint* places a restriction on the manufactured product amount or the production time

- for each product, the manufactured amount of the product is as least as high as that product's lower production limit, but does not exceed that product's upper production limit:
  - $\forall p \in PRODUCT, minTons(p) \leq Tons(p) \leq maxTons(p)$
- the total production time does not exceed the available factory time:
  - $\sum_{p \in PRODUCT} \frac{1}{tonsPerHr(p)} \leq maxHrs$

Installed Program

The entity and the parameters of the problem are defined as:

```
//sets
PRODUCT(x), PRODUCT:name(x:n) -> string(n).

//parameters
tonsPerHr[p]=v -> PRODUCT(p), float[64](v).
hrsPerTon[p]=v -> PRODUCT(p), float[64](v).
hrsPerTon[p]=v <- v=1/tonsPerHr[p].
profitPerTon[p]=v -> PRODUCT(p), float[64](v).
```

```
minTons[p]=v -> PRODUCT(p), float[64](v).
maxTons[p]=v -> PRODUCT(p), float[64](v).

//constants
maxHrs[]=v -> float[64](v).
+maxHrs[]=40.
```

The variable is indexed by the products:

```
Tons[p]=v -> PRODUCT(p), float[64](v).
PRODUCT(p) -> Tons[p]=_ .
lang:solver:variable(`Tons).
```

The goal of the computation—to maximize the objective function $\sum_{p \in PRODUCT} Tons(p) * profitPerTon(p)$—is encoded as:

```
lang:solver:maximal(`TotalProfit).
TotalProfit[]=v -> float[64](v).
TotalProfit[]+=Tons[p]*profitPerTon[p].
```

The constraint $\forall p \in PRODUCT, minTons(p) \leq Tons(p)$ is encoded as:

```
PRODUCT(p) -> Tons[p]>=minTons[p].
```

The constraint $\forall p \in PRODUCT, Tons(p) \leq maxTons(p)$ is encoded as:

```
PRODUCT(p) -> Tons[p]<=maxTons[p].
```

The constraint $\sum_{p \in PRODUCT} \frac{1}{tonsPerHr(p)} \leq maxHrs$ is encoded as:

```
totalProductHrs[]=v -> float[64](v).
totalProductHrs[]+=Tons[p]*hrsPerTon[p].
totalProductHrs[]=v1,maxHrs[]=v2 -> v1<=v2.
```

Execution

The program is executed for given index sets along with their associated parameters, for example:

```
+PRODUCT(x), +PRODUCT:name(x:"bands").
+PRODUCT(x), +PRODUCT:name(x:"coils").
+PRODUCT(x), +PRODUCT:name(x:"plate").

+tonsPerHr[p] = 200 <- PRODUCT:name(p:"bands").
+tonsPerHr[p] = 140 <- PRODUCT:name(p:"coils").
+tonsPerHr[p] = 160 <- PRODUCT:name(p:"plate").

+profitPerTon[p] = 25 <- PRODUCT:name(p:"bands").
+profitPerTon[p] = 30 <- PRODUCT:name(p:"coils").
+profitPerTon[p] = 29 <- PRODUCT:name(p:"plate").

+maxTons[p] = 6000 <- PRODUCT:name(p:"bands").
+maxTons[p] = 4000 <- PRODUCT:name(p:"coils").
+maxTons[p] = 3500 <- PRODUCT:name(p:"plate").

+minTons[p] = 1000 <- PRODUCT:name(p:"bands").
+minTons[p] = 500 <- PRODUCT:name(p:"coils").
+minTons[p] = 750 <- PRODUCT:name(p:"plate").
```

```
+Tons_expected[p] = 6000 <- PRODUCT:name(p:"bands").
+Tons_expected[p] = 500 <- PRODUCT:name(p:"coils").
+Tons_expected[p] = 1028.57142857143 <- PRODUCT:name(p:"plate").
```

The solver computes the values for the variable `Tons[p]`. The result is tested by:

```
+system:bloxunit:Compare:expected[c] = "Tons_expected",
+system:bloxunit:Compare:actual[c] = "Tons" <- system:bloxunit:Compare:id(c:0).
```

3.5 Production Problem (version 3, suiteProductionStages)

Scenario: Ben further refines his factory's work scheme by splitting the overall available factory time into stages. Now, the production rate for a product may be different for different stages. Also, each stage is associated with a maximum available time.

Problem Specification:

Each entity represents a set indexing the parameter and variable predicates:

- *PRODUCT*—set of products manufactured at the factory
- *STAGE*—set of production stages

Each *parameter* stores some externally provided information:

- *tonsPerHr(p,s)*—number of tons of product *p* that can be manufactured in one hour during state *s*
- *profitPerTon(p)*—profit provided by one ton of product *p*
- *minTons(p)*—the lower production limit for product *p*
- *maxTons(p)*—the upper production limit for product *p*
- *maxHrs(s)*—number of available factory hours at stage *s*

The *variable Tons(p)* represents the number of tons of product *p* to be manufactured

The *objective function* defines the total profit generated at the factory

- $\sum_{p \in PRODUCT} Tons(p) * profitPerTon(p)$

Each *constraint* places a restriction on the manufactured product amount or the production time

- for each product, the manufactured amount of the product is as least as high as that product's lower production limit, but does not exceed that product's upper production limit:
  - $\forall p \in PRODUCT, minTons(p) \leq Tons(p) \leq maxTons(p)$
- the total production time does not exceed the available factory time:
  - $\forall s \in STAGES, \sum_{p \in PRODUCT} \frac{1}{tonsPerHr(p,s)} \leq maxHrs(s)$

Installed Program

The entities and parameters of the problem are defined as:

```
//sets
PRODUCT(x), PRODUCT:name(x:n) -> string(n).
STAGE(x), STAGE:name(x:n) -> string(n).

//parameters
```

```
maxHrs[s]=v -> STAGE(s), float[64](v), v>=0.
tonsPerHr[p,s]=v -> PRODUCT(p), STAGE(s), float[64](v), v>=0.
hrsPerTon[p,s]=v -> PRODUCT(p), STAGE(s), float[64](v).
hrsPerTon[p,s]=v <- v=1/tonsPerHr[p,s].
profitPerTon[p]=v -> PRODUCT(p), float[64](v).
maxTons[p]=v -> PRODUCT(p), float[64](v), v>=0.
minTons[p]=v -> PRODUCT(p), float[64](v), v>=0.
```

The variable is indexed by the products:

```
Tons[p]=v -> PRODUCT(p), float[64](v).
PRODUCT(p) -> Tons[p]=_.
lang:solver:variable(`Tons).
```

The goal of the computation—to maximize the objective function $\sum_{p \in PRODUCT} Tons(p) * profitPerTon(p)$—is encoded as:

```
lang:solver:maximal(`TotalProfit).
TotalProfit[]=v -> float[64](v).
TotalProfit[]+=Tons[p]*profitPerTon[p].
```

The constraint $\forall p \in PRODUCT, minTons(p) \leq Tons(p)$ is encoded as:

```
PRODUCT(p) -> Tons[p]>=minTons[p].
```

The constraint $\forall p \in PRODUCT, Tons(p) \leq maxTons(p)$ is encoded as:

```
PRODUCT(p) -> Tons[p]<=maxTons[p].
```

The constraint $\forall s \in STAGES, \sum_{p \in PRODUCT} \frac{1}{tonsPerHr(p,s)} \leq maxHrs(s)$ is encoded as:

```
totalProductHrs[s]=v -> STAGE(s), float[64](v).
totalProductHrs[s]+=Tons[p]*hrsPerTon[p,s].
totalProductHrs[s]=v1, maxHrs[s]=v2 -> v1<=v2.
```

Execution

The program is executed for given index sets along with their associated parameters, for example:

```
+PRODUCT(x), +PRODUCT:name(x:"bands").
+PRODUCT(x), +PRODUCT:name(x:"coils").
+PRODUCT(x), +PRODUCT:name(x:"plate").

+STAGE(s), +STAGE:name(s:"reheat").
+STAGE(s), +STAGE:name(s:"roll").

+maxHrs[s] = 35 <- STAGE:name(s:"reheat").
+maxHrs[s] = 40 <- STAGE:name(s:"roll").

+tonsPerHr[p, s] = a <- PRODUCT:name(p:"bands"), STAGE:name(s:"reheat"), a=200.
+tonsPerHr[p, s] = a <- PRODUCT:name(p:"coils"), STAGE:name(s:"reheat"), a=200.
+tonsPerHr[p, s] = a <- PRODUCT:name(p:"plate"), STAGE:name(s:"reheat"), a=200.
+tonsPerHr[p, s] = a <- PRODUCT:name(p:"bands"), STAGE:name(s:"roll"), a=200.
+tonsPerHr[p, s] = a <- PRODUCT:name(p:"coils"), STAGE:name(s:"roll"), a=140.
+tonsPerHr[p, s] = a <- PRODUCT:name(p:"plate"), STAGE:name(s:"roll"), a=160.

+profitPerTon[p] = 25 <- PRODUCT:name(p:"bands").
```

```
+profitPerTon[p] = 30 <- PRODUCT:name(p:"coils").
+profitPerTon[p] = 29 <- PRODUCT:name(p:"plate").

+maxTons[p] = 6000 <- PRODUCT:name(p:"bands").
+maxTons[p] = 4000 <- PRODUCT:name(p:"coils").
+maxTons[p] = 3500 <- PRODUCT:name(p:"plate").
+minTons[p] = 1000 <- PRODUCT:name(p:"bands").
+minTons[p] = 500 <- PRODUCT:name(p:"coils").
+minTons[p] = 750 <- PRODUCT:name(p:"plate").

+Tons_expected[p] = 3357.14285714286 <- PRODUCT:name(p:"bands").
+Tons_expected[p] = 500 <- PRODUCT:name(p:"coils").
+Tons_expected[p] = 3142.85714285714 <- PRODUCT:name(p:"plate").
```

The solver computes the values for the variable `Tons[p]`. The result is tested by:

```
+system:bloxunit:Compare:expected[c] = "Tons_expected",
+system:bloxunit:Compare:actual[c] = "Tons" <- system:bloxunit:Compare:id(c:0).
```

3.6 Transportation Problem (suiteTransportation)

Scenario: Andy owns a transportation company shipping lumber between a set of origin and destination locations. Each origin location offers certain supply of lumber, and each destination location specifies certain demand for lumber. Furthermore, each pair of origin-destination locations has associated cost of lumber shipping. Andy's concern is to minimize the total cost of shipping lumber, at the same time making sure that (1) the total amount of lumber shipped out of each origin location equals that location's supply, and (2) the total amount of lumber delivered to each destination location equals that location's demand.

Problem Specification:

Each entity represents a set indexing the parameter and variable predicates:

- *ORIG*—set of origin locations
- *DEST*—set of destination locations

Each *parameter* stores some externally provided information:

- *supply(o)*—supply of lumber at origin location *o*
- *demand(d)*—demand for lumber at destination location *d*
- *cost(o,d)*—cost of shipping lumber from origin location *o* to destination location *d*

The *variable Trans(o,d)* represents the number of tons of lumber shipped from origin location *o* to destination location *d*

The *objective function* defines the total cost of shipping the lumber

- $\sum_{o \in ORIG} \sum_{d \in DEST} Trans(o,d) * cost(o,d)$

Each *constraint* places a restriction on the amount of lumber being shipped:

- for each origin location, the sum of all lumber shipments out of the location is equal to the supply available at that location
  - $\forall o \in ORIG, \sum_{d \in DEST} Trans(o,d) = supply(o)$
- for each destination location, the sum of all lumber deliveries to the location is equal to the demand at that location

- $\forall d \in DEST, \sum_{o \in ORIG} Trans(o,d) = demand(d)$

Installed Program

The entities and parameters of the problem are defined as:

```
// sets
ORIG(o), ORIG:name(o:n) -> string(n).
DEST(d), DEST:name(d:n) -> string(n).

// parameters
supply[o]=v -> ORIG(o), float[64](v), v>0.
demand[d]=v -> DEST(d), float[64](v), v>0.
cost[o,d]=v -> ORIG(o), DEST(d), float[64](v), v>=0.
```

The variable is indexed by the origin and destination locations:

```
Trans[o,d]=v -> ORIG(o), DEST(d), float[64](v), v>=0.
ORIG(o), DEST(d) -> Trans[o,d]=_ .
lang:solver:variable('Trans).
```

The goal of the computation—to minimize the objective function $\sum_{o \in ORIG} \sum_{d \in DEST} Trans(o,d) * cost(o,d)$—is encoded as:

```
lang:solver:minimal('TotalCost).
TotalCost[]=v -> float[64](v).
TotalCost[]+=Trans[o,d]*cost[o,d].
```

The constraint $\forall o \in ORIG, \sum_{d \in DEST} Trans(o,d) = supply(o)$ is encoded as:

```
totalShipmentFrom[o]=v -> ORIG(o), float[64](v).
totalShipmentFrom[o]+=Trans[o,_].
ORIG(o), totalShipmentFrom[o]=v1, supply[o]=v2 -> v1=v2.
```

The constraint $\forall d \in DEST, \sum_{o \in ORIG} Trans(o,d) = demand(d)$ is encoded as:

```
totalShipmentTo[d]=v -> DEST(d), float[64](v).
totalShipmentTo[d]+=Trans[_,d].
DEST(d), totalShipmentTo[d]=v1, demand[d]=v2 -> v1=v2.
```

Execution

To solve the problem, we load the above program block, and execute the *test.init* block providing the values for index sets and parameters. The solver computes the values for the variable *Trans[o,d]*. The result is tested by the *test.assert* block.

3.7 Transportation Problem (version 2, suiteTransportationMulti-Product)

Scenario: Andy decides to expand the range of his company's services by offering shipments for a variety of products. Consequently, each origin location provides certain supply of each product, and each destination location specifies certain demand for each product. Also, each pair of origin-destination locations has associated shipping cost for each product. Furthermore, there is a limit of shipments that can occur between each pair of origin and destination locations. Now, Andy's concern is to minimize the total cost of shipping all products, at the same time making sure that (1) for each product, the total amount of the product shipped our of each origin location equals that location's supply of

that product, (2) for each product, the total amount of the product delivered to each destination location equals that location's demand for that product, and (3) the total amount of products shipped between each pair of origin and destination locations does not exceed the shipment limit for this location pair.

Problem Specification:

Each entity represents a set indexing the parameter and variable predicates:

- *ORIG*—set of origin locations
- *DEST*—set of destination locations
- *PRODUCT*—set of products

Each *parameter* stores some externally provided information:

- *supply(o,p)*—supply of product *p* at origin location *o*
- *demand(d,p)*—demand for product *p* at destination location *d*
- *cost(o,d,p)*—cost of shipping product *p* from origin location *o* to destination location *d*
- *limit(o,d)—limit of shipments between origin location *o* and destination location *d*

The *variable Trans(o,d,p)* represents the number of tons of product p shipped from origin location o to destination location d.

The *objective function* defines the total cost of shipping all products:

- $\sum_{o \in ORIG} \sum_{d \in DEST} \sum_{p \in PRODUCT} Trans(o,d,p) * cost(o,d,p)$

Each *constraint* places a restriction on the amount of products being shipped:

- for each origin location, for each product, the sum of all shipments of the product out of the location is equal to the supply of that product available at that location
  - $\forall o, \in ORIG, \forall p \in PRODUCT, \sum_{d \in DEST} Trans(o,d,p) = supply(o,p)$
- for each destination location, for each product, the sum of all deliveries of the product to the location is equal to the demand for that product at that location
  - $\forall d, \in DEST, \forall p \in PRODUCT, \sum_{o \in ORIG} Trans(o,d,p) = demand(d,p)$
- for reach origin location, for each destination location, the sum of all shipments from the origin location to the destination location is not larger than the shipment limit for these location pair
- $\forall o \in ORIG, \forall d \in DEST, \sum_{p \in PRODUCT} Trans(o,d,p) \leq limit(o,d)$

Installed Program

The entities and parameters of the problem are defined as:

```
// sets
ORIG(o), ORIG:name(o:n) -> string(n).
DEST(d), DEST:name(d:n) -> string(n).
PRODUCT(p), PRODUCT:name(p:n) -> string(n).

// parameters
supply[o,p]=v -> ORIG(o), PRODUCT(p), float[64](v), v>=0.
demand[d,p]=v -> DEST(d), PRODUCT(p), float[64](v), v>=0.
limit[o,d]=v -> ORIG(o), DEST(d), float[64](v), v>=0.
cost[o,d,p]=v -> ORIG(o), DEST(d), PRODUCT(p), float[64](v), v>=0.
```

The variable is indexed by the origin and destination locations and the products:

```
Trans[o,d,p]=v -> ORIG(o), DEST(d), PRODUCT(p), float[64](v), v>=0.
ORIG(o), DEST(d), PRODUCT(p) -> Trans[o,d,p]=_ .
lang:solver:variable('Trans).
```

The goal of the computation—to minimize the objective function $\sum_{o \in ORIG} \sum_{d \in DEST} \sum_{p \in PRODUCT} Trans(o,d,p) * cost(o,d,p)$—is encoded as:

```
lang:solver:minimal('TotalCost).
TotalCost[]=v -> float[64](v).
TotalCost[]+=Trans[o,d,p]*cost[o,d,p].
```

The constraint $\forall o \in ORIG, \forall p \in PRODUCT, \sum_{d \in DEST} Trans(o,d,p) = supply(o,p)$ is encoded as:

```
totalShipmentsFrom[o,p]=v -> ORIG(o), PRODUCT(p), float[64](v).
totalShipmentsFrom[o,p]+=Trans[o,_,p].
ORIG(o), PRODUCT(p), totalShipmentsFrom[o,p]=v1, supply[o,p]=v2 -> v1=v2.
```

The constraint $\forall d \in DEST, \forall p \in PRODUCT, \sum_{o \in ORIG} Trans(o,d,p) = demand(d,p)$ is encoded as:

```
totalShipmentsTo[d,p]=v -> DEST(d), PRODUCT(p), float[64](v).
totalShipmentsTo[d,p]+=Trans[_,d,p].
DEST(d), PRODUCT(p), totalShipmentsTo[d,p]=v1, demand[d,p]=v2 -> v1=v2.
```

The constraint $\forall o \in ORIG, \forall d \in DEST, \sum_{p \in PRODUCT} Trans(o,d,p) \leq limit(o,d)$ is encoded as:

```
totalShipmentsFromTo[o,d]=v -> ORIG(o), DEST(d), float[64](v).
totalShipmentsFromTo[o,d]+=Trans[o,d,_].
ORIG(o), DEST(d), totalShipmentsFromTo[o,d]=v1, limit[o,d]=v2 -> v1<=v2.
```

Execution

To solve the problem, we load the above program block, and execute the *test.init* block providing the values for index sets and parameters. The solver computes the values for the variable *Trans[o,d,p]*. The result is tested by the *test.assert* block.

3.8 Transportation Problem (version 3, suiteTransportationLink)

Scenario: Since product shipments occur only between certain pairs of origin and destination locations (rather than between every possible origin-destination combination), to improve the efficiency of the search, Andy identifies the location pairs with product shipments as links. We test this new feature using the lumber shipping scenario. The objective now is to minimize the total cost of shipping lumber on all links, at the same time making sure that, for each link, (1) the total amount of lumber shipped our of the link's origin location equals that location's supply, and (2) the total amount of lumber delivered to the link's destination location equals that location's demand. .

Problem Specification:

Each entity represents a set indexing the parameter and variable predicates:

- *ORIG*—set of origin locations
- *DEST*—set of destination locations
- *LINK*—set of origin-destination location pairs such that the lumber is shipped from the link's origin location to the link's destination location

Each *parameter* stores some externally provided information:

- *supply(o)*—supply of lumber at origin location o
- *demand(d)*—demand for lumber at destination location d
- *cost(o,d)*—cost of shipping lumber between linked locations o and d

The *variable Trans(o,d)* represents the number of tons of lumber shipped from origin location o to destination location d.

The *objective function* defines the total cost of shipping all products:

- $\sum_{o,d \in LINK} Trans(o,d) * cost(o,d)$

Each *constraint* places a restriction on the amount of lumber being shipped:

- for each origin location, the sum of all lumber shipments out of the location is equal to the lumber supply available at that location
  - $\forall o, \in ORIG, \sum_{d \in DEST} Trans(o,d) = supply(o)$
- for each destination location, the sum of all lumber deliveries to the location is equal to the demand for lumber at that location
  - $\forall d, \in DEST, \sum_{o \in ORIG} Trans(o,d) = demand(d,)$

Installed Program

The entities and parameters of the problem are defined as:

```
//sets
ORIG(o), ORIG:name(o:n) -> string(n).
DEST(d), DEST:name(d:n) -> string(n).
LINK(l), LINK:id(l:i) -> int[32](i).
LINK:orig[l]=o -> LINK(l), ORIG(o), lang:inv[LINK:orig][o]=l.
LINK:dest[l]=d -> LINK(l), DEST(d), lang:inv[LINK:dest][d]=l.

//parameters
supply[o]=v -> ORIG(o), float[64](v), v>=0.
demand[d]=v -> DEST(d), float[64](v), v>=0.
cost[l]=v -> LINK(l), float[64](v), v>=0.
```

Note that the index set *LINK* is defined over two other index sets, *ORIG* and *DEST*

The variable is indexed by the linked origin and destination locations:

```
Trans[l]=v -> LINK(l), float[64](v), v>=0.
LINK(l) -> Trans[l]=_.
lang:solver:variable('Trans).
```

The goal of the computation—to minimize the objective function $\sum_{o,d \in LINK} Trans(o,d) * cost(o,d)$—is encoded as:

```
lang:solver:minimal('TotalCost).
TotalCost[]=v -> float[64](v).
TotalCost[]+=Trans[l]*cost[l].
```

The constraint $\forall o \in ORIG, \sum_{d \in DEST} Trans(o,d) = supply(o)$ is encoded as:

```
fromLink(o,l) -> ORIG(o), LINK(l).
fromLink(o,l) <- ORIG(o), LINK:orig[l]=o.
totalShipmentFrom[o]=v -> ORIG(o), float[64](v).
```

```
totalShipmentFrom[o]=v <- agg<<v=total(r)>> fromLink(o,l), Trans[l]=r.
ORIG(o), totalShipmentFrom[o]=v1, supply[o]=v2 -> v1=v2.
```

The constraint $\forall d \in DEST, \sum_{o \in ORIG} Trans(o,d) = demand(d)$ is encoded as:

```
toLink(d,l) -> DEST(d), LINK(l).
toLink(d,l) <- LINK:dest[l]=d.
totalShipmentTo[d]=v -> DEST(d), float[64](v).
totalShipmentTo[d]=v <- agg<<v=total(r)>> toLink(d,l), Trans[l]=r.
DEST(d), totalShipmentTo[d]=v1, demand[d]=v2 -> v1=v2.
```

Execution

To solve the problem, we load the above program block, and execute the *test.init* block providing the values for index sets and parameters. The solver computes the values for the variable *Trans[o,d]*. The result is tested by the *test.assert* block.

3.9 Production-Transportation Problem (suiteProductionTransformation)

Scenario: Ben and Andy decide to combine efforts and form a unified production-transportation venture B&A. B&A spans over a set of origin and destination locations, and handles a range of products. Product factories (at origin locations) are parameterized by the amount of each product that can be manufactured in certain amount of time, the cost of manufacturing certain amount of each product, and the available factory time. Each destination location specifies certain demand for each product. Furthermore, each pair of origin-destination locations has associated shipping cost for each product. The goal of Ben and Andy is to minimize the total cost of manufacturing and shipping the products, at the same time making sure that (1) for each product, the total amount of the product shipped our of each origin location equals that location's supply for that product, (2) for each product, the total amount of the product delivered to each destination location equals that location's demand for that product, and (3) total production at each factory does not exceed that factory's available time.

Problem Specification:

Each entity represents a set indexing the parameter and variable predicates:

- *ORIG*—set of origin locations
- *DEST*—set of destination locations
- *PRODUCT*—set of products

Each *parameter* stores some externally provided information:

- *tonsPerHr(o,p)*—number of tons of product *p* that can be manufactured in one hour in the factory at location o
- *maxHrs(o)*—number of hours available at the factory at location o
- *demand(d,p)*—demand for product *p* at destination location *d*
- *makeCost(o,p)*—cost of manufacturing product *p* at the factory at location *o*
- *transCost(o,d,p)*—cost of shipping product *p* from origin location *o* to destination location *d*

Each *variable* represents a value to be computed:

- *Make(o,p)*—the number of tons of product *p* to be manufactured at the factory at location *o*
- *Trans(o,d,p)*—the number of tons of product *p* shipped from origin location *o* to destination location *d*

The *objective function* defines the total cost of manufacturing and shipping all products:

- $\sum_{o \in ORIG} \sum_{p \in PRODUCT} makeCost(o,p) * Make(o,p) + \sum_{o \in ORIG} \sum_{d \in DEST} \sum_{p \in PRODUCT} transCost(o,d,p) * Trans(o,d,p)$

Note that the objective function involves two variables.

Each *constraint* places a restriction on the amount of products being manufactured or shipped, or on the production time:

- for each origin location, for each product, the sum of all shipments of the product out of the location is equal to the supply of that product available at that location
  - $\forall o \in ORIG, \forall p \in PRODUCT, \sum_{d \in DEST} Trans(o,d,p) = supply(o,p)$
- for each destination location, for each product, the sum of all deliveries of the product to the location is equal to the demand for that product at that location
  - $\forall d \in DEST, \forall p \in PRODUCT, \sum_{o \in ORIG} Trans(o,d,p) = demand(d,p)$
- for each origin location, the total production time at the location does not exceed the factory time available at that location:
  - $\forall o \in ORIG, \sum_{p \in PRODUCT} \frac{1}{tonsPerHr(p)} \leq maxHrs(o)$

Installed Program

The entities and parameters of the problem are defined as:

```
// sets
ORIG(o), ORIG:name(o:n) -> string(n).
DEST(d), DEST:name(d:n) -> string(n).
PRODUCT(p), PRODUCT:name(p:n) -> string(n).

// parameters
tonsPerHr[o,p]=v -> ORIG(o), PRODUCT(p), float[64](v),v>=0.
hrsPerTon[o,p]=v -> ORIG(o), PRODUCT(p), float[64](v).
hrsPerTon[o,p]=r <- (r=1/tonsPerHr[o,p]).
maxHrs[o]=v -> ORIG(o), float[64](v), v>=0.
demand[d, p]=v -> DEST(d), PRODUCT(p), float[64](v), v>=0.
makeCost[o,p]=v -> ORIG(o), PRODUCT(p),  float[64](v), v>=0.
transCost[o,d,p]=v -> ORIG(o), DEST(d), PRODUCT(p),  float[64](v), v>=0.
```

The problem involves two variables:

```
Make[o,p]=v -> ORIG(o), PRODUCT(p), float[64](v), v>=0.
ORIG(o), PRODUCT(p) -> Make[o,p]=_.

Trans[o,d,p]=v -> ORIG(o), DEST(d), PRODUCT(p), float[64](v), v>=0.
ORIG(o), DEST(d), PRODUCT(p) -> Trans[o,d,p]=_ .

lang:solver:variable(`Make).
lang:solver:variable(`Trans).
```

The goal of the computation—to minimize the objective function:

$\sum_{o \in ORIG} \sum_{p \in PRODUCT} makeCost(o,p) * Make(o,p)$
$+ \sum_{o \in ORIG} \sum_{d \in DEST} \sum_{p \in PRODUCT} transCost(o,d,p) * Trans(o,d,p)$

is encoded as:

```
lang:solver:minimal('TotalCost).
TotalCost[]=v -> float[64](v).
TotalCost[] = TotalMakeCost[]+TotalTransCost[].

TotalMakeCost[]=v -> float[64](v).
TotalMakeCost[]+=makeCost[o,p]*Make[o,p].

TotalTransCost[]=v -> float[64](v).
TotalTransCost[]+=transCost[o,d,p]*Trans[o,d,p].
```

The constraint $\forall o \in ORIG, \forall p \in PRODUCT, \sum_{d \in DEST} Trans(o,d,p) = supply(o,p)$ is encoded as:

```
totalShipOut[o,p]=v -> ORIG(o), PRODUCT(p), float[64](v).
totalShipOut[o,p]+=Trans[o,_,p].
ORIG(o),PRODUCT(p),totalShipOut[o,p]=v1,Make[o,p]=v2,v2-v1=x-> x=0.
```

The constraint $\forall d \in DEST, \forall p \in PRODUCT, \sum_{o \in ORIG} Trans(o,d,p) = demand(d,p)$ is encoded as:

```
totalDelivery[d,p]=v -> DEST(d), PRODUCT(p), float[64](v).
totalDelivery[d,p]+=Trans[_,d,p].
DEST(d), PRODUCT(p), totalDelivery[d,p]=v1, demand[d,p]=v2 -> v1=v2.
```

The constraint $\forall o \in ORIG, \sum_{p \in PRODUCT} \frac{1}{tonsPerHr(p)} \leq maxHrs(o)$ is encoded as:

```
totalProductHrs[o]=v -> ORIG(o), float[64](v).
totalProductHrs[o]+=hrsPerTon[o,p]*Make[o,p].
totalProductHrs[o]=v1, maxHrs[o]=v2 -> v1<=v2.
```

Execution

To solve the problem, we load the above program block, and execute the *test.init* block providing the values for index sets and parameters. The solver computes the values for the variables *Make[o,p]* and *Trans[o,d,p]*. The result is tested by the *test.assert* block.

3.10 Scheduling Problem (suiteScheduling)

Scenario: Paula is a nurse manager in a large hospital. The nurses' work time is split into several shifts, (e.g., Shift1=Mon/AM, Shift2=Mon/PM, Shift3=Mon/Eve etc.). Each shift has a minimum number of nurses required to work at that shift. Sets of shifts form schedules (e.g., Schedule1 = {Mon/AM, Tue/PM, Wed/AM, Thu/PM, Fri/AM}), and each schedule has an associated per-person pay rate. Paula's task is to schedule nurses' work in a way that minimizes the total work cost, and at the same time satisfies the minimum personnel constraint.

Problem Specification:

Each entity represents a set indexing the parameter and variable predicates:

- *SHIFTS*—set of work shifts
- *SCHEDS*—set of schedules

Each *parameter* stores some externally provided information:

- *shiftList(i,j)*—assignment of shift *i* to schedule *j* (*1* if shift *i* is in schedule *j*, *0* otherwise)
- *shiftReq(i)*—minimum number of people required to work at shift *i*
- *rate(j)*—per-person pay rate on schedule *j*

The *variable Work(j)* represents the number of people assigned to work at schedule *j*.

The *objective function* defines the total cost of scheduled work:

- $\sum_{j \in SCHEDS} rate(j) * Work(j)$

For each shift, the *constraint* requires that the total number of people assigned to work at that shift at different schedules is not lower than the minimum number of people required to work at that shift

- $\forall i \in SHIFTS \sum_{j \in SCHEDS\ s.t.\ shiftList[i,j]=1} Work(j) \geq shiftReq()$

Note that the constraint requires filtering the values of the *SCHEDS* index set

Installed Program

The parameters of the problem are defined as:

```
//sets
SHIFTS(i), SHIFTS:name(i:n) -> string(n).
SCHEDS(j), SCHEDS:id(j:n) -> int[32](n).

//parameters
shiftList[i,j]=a -> SHIFTS(i), SCHEDS(j), int[32](a), a>=0, a<=1.
shiftReq[i]=b -> SHIFTS(i), int[32](b), b>=0.
rate[j]=c -> SCHEDS(j), int[32](c), c>=0.
```

The variable is indexed by food products:

```
Work[j]=w -> SCHEDS(j), int[32](w), w>=0.
SCHEDS(j) -> Work[j]=_.
lang:solver:variable(`Work).
```

The goal of the computation—to minimize the objective function $\sum_{j \in SCHEDS} rate(j) * Work(j)$—is encoded as:

```
lang:solver:minimal(`TotalCost).
TotalCost[]=v -> int[32](v).
TotalCost[]+=rate[j]*Work[j].
```

The constraint $\forall i \in SHIFTS \sum_{j \in SCHEDS\ s.t.\ shiftList[i,j]=1} Work(j) \geq shiftReq()$ is encoded as:

```
shift_sched(i,j) -> SHIFTS(i), SCHEDS(j).
shift_sched(i,j) <- shiftList[i,j]=1.
totalShiftWork[i]=v -> SHIFTS(i), int[32](v).
totalShiftWork[i]=v <- agg<<v=total(r)>> shift_sched(i,j), Work[j]=r.
SHIFTS(i) -> totalShiftWork[i]>=shiftReq[i].
```

Execution

To solve the problem, we load the above program block, and execute the *testA.init* block providing the values for index sets and parameters. The solver computes the values for the variable *Work[j]*. The result is tested by the *testA.assert* block.

3.11 Scheduling Problem—execution with multiple data sets

We have executed the program encoding the scheduling problem over two distinct data sets. Doing so required cleaning up the database after the first data set run. For that purpose, we added an auxiliary test file *testA.cleanup*, removing the contents of the index set predicates using delta rules:

```
-SHIFTS(i)<-SHIFTS@previous(i).
-SCHEDS(j)<-SCHEDS@previous(j).
```

The effects of the auxiliary test are checked by the assertion that requires the index set predicates to be empty:

```
!SHIFTS(_).
!SCHEDS(_).
```

After successful termination of the auxiliary test, we could run the program again by executing the *testB.init* block providing another data set. Consequently, the solver computed a new set of values for the variable *Work[j]*. The result was tested by *testB.assert*.

3.12 PDX Assortment (part 1, suitePDXAssortment1)

This section presents a real-life example from a Predictix Assortment application which solves an assortment-with-space problem. The goal is to determine the optimal subset of products to be carried in each store class, at the same satisfying physical store constraints (such as the size of the shelf) and, if possible, vendor-specified constraints limiting how much of a given sub-category of a product should be carried.

The problem involves the following entities:

- a set of *products*.
- a set of *pogs* (or plan-o-grams), which specify the physical characteristic of a set of stores
- a set of *rules*, some of which are *at-least rules*, and some of which are *at-most rules*

The parameters are:

- a business value for each product
- a business value for each rule
- a function specifying how much a product participates in satisfying or breaking a rule
- the limit specified by each rule
- the amount of space (width) used by each item on the shelf

We encode this problem as two linear DatalogLB programs. The first program finds the most valuable set of at-least rules which can be satisfied. Its result is then used by the second program to determine the optimal subset of products which can be picked to satisfy those rules.

Installed program

The entities and parameters for the first program are defined as:

```
// sets
item(x),item:product_id[x]=s -> string(s), lang:inv[item:product_id][s]=x.
item:value[i]=n -> item(i), int[32](n).

rules:rule(r),rules:rule:name[r]=n -> string(n), lang:inv[rules:rule:name][n]=r.
rules:rule:at_least_rule(r) -> rules:rule(r).
rules:rule:at_most_rule(r) -> rules:rule(r).
rules:rule:value[r]=x -> rules:rule(r), int[32](x).
rules:rule:quantity[r]=x -> rules:rule(r), int[32](x).

pog(p),pog:name[p]=n -> string(n), lang:inv[pog:name][n]=p.
```

```
// parameters
potential[item,rule,pog]=n -> item(item), rules:rule(rule), pog(pog), int[32](n).
width[i,p]=n -> pog(p), item(i), int[32](n).
shelf_width[p]=x -> pog(p), int[32](x).
```

The program defines one variable per at-least rule and per pog, and one variable per item and per pog:

```
satisfied[r,p]=s -> pog(p), rules:rule:at_least_rule(r), int[32](s), s>=0, s<=1.
pog(p), rules:rule:at_least_rule(r) -> satisfied[r,p]=_ .

needed[i,p]=n -> pog(p), item(i), int[32](n), n>=0, n<=1.
pog(p), item(i) -> needed[i,p]=_.

lang:solver:variable(`needed).
lang:solver:variable(`satisfied).
```

The constraints are for each (rule,pog) pair:

```
total_achieved[r,p]=t -> pog(p), rules:rule(r), int[32](t).
total_achieved[r,p]+=potential[i,r,p]*needed[i,p].
pog(p),rules:rule(r) -> total_achieved[r,p]-rules:rule:quantity[r]*satisfied[r,p]<=0.
```

and for each pog for the space constraint:

```
filled_shelf[p]=f -> pog(p), int[32](f).
filled_shelf[p]+=width[i,p]*needed[i,p].
pog(p) -> filled_shelf[p]<=shelf_width[p].
```

The objective function is:

```
lang:solver:maximal(`objective).
objective[]=v -> int[32](v).
objective[]+=satisfied[r,p]*rules:rule:value[r].
```

The problem is solved in the same manner as the problems presented in the previous sections. Once the values of the variables are computed, we use them as parameters for the second part of the problem specification described in Section *PDX Assortment (part 2, suitePDXAssortment2)*.

3.13 PDX Assortment (part 2, suitePDXAssortment2)

The second part of the assortment problem encoding uses the result of the first program—which finds the optimal subset of satisfiable rules—to determine the optimal subset of products which can be picked to satisfy those rules.

In the current state of BloxOptimize, all predicates used in the definition of a linear program have to be declared in that program, so the entities and predicates from the previous program are re-declared:

```
// sets
item(x),item:product_id[x]=s -> string(s), lang:inv[item:product_id][s]=x.
item:value[i]=n -> item(i), int[32](n).

rules:rule(r),rules:rule:name[r]=n -> string(n), lang:inv[rules:rule:name][n]=r.
rules:rule:quantity[r]=x -> rules:rule(r), int[32](x).
rules:rule:value[r]=x -> rules:rule(r), int[32](x).
rules:rule:at_least_rule(r) -> rules:rule(r).
```

```
rules:rule:at_most_rule(r) -> rules:rule(r).

// parameters
potential[item,rule,pog]=n -> item(item), rules:rule(rule), pog(pog), int[32](n).
width[i,p]=n -> pog(p), item(i), int[32](n).
shelf_width[p]=x -> pog(p), int[32](x).
```

One of the parameters depends on a predicate which itself depends on a variable from the previous program:

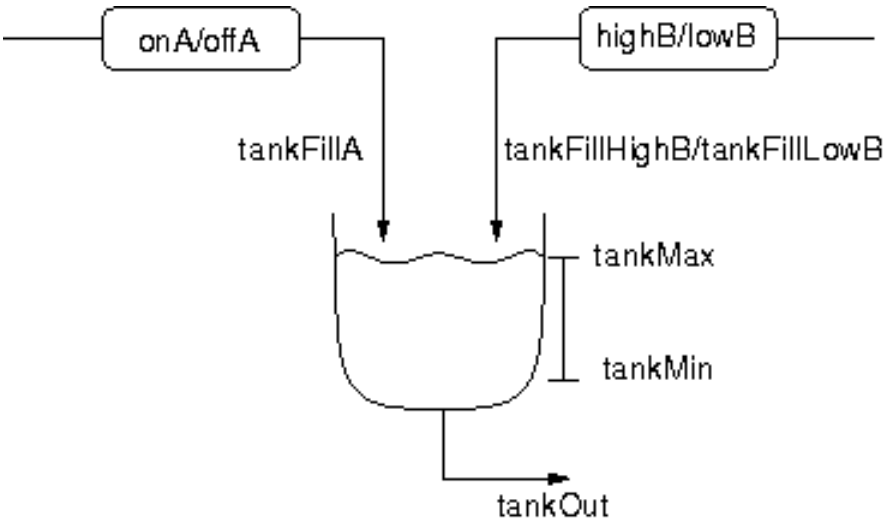
```
rules:rule:satisfiable_rule(r,p) <-
                            rules:rule:at_most_rule(r),
                            total_achieved[r,p]=s,
                            rules:rule:quantity[r]=l, s<l.

rules:rule:satisfiable_rule2(r,p) <-
                             rules:rule:at_least_rule(r),
                             total_achieved[r,p]=s,
                             rules:rule:quantity[r]=l, s>l.
```

The rest of the program is defined as expected:

```
// variables
picked[i,p]=n -> item(i), pog(p), int[32](n), n>=0, n<=1.
item(i), pog(p) -> picked[i,p]=_ .
lang:solver:variable(`picked).
// objective function
lang:solver:maximal(`objective2).
objective2[]=s -> int[32](s).
objective2[]+=picked[i,p]*item:value[i].

// constraints
rules:rule:satisfiable_rule(r,p) -> pog(p), rules:rule(r).
rules:rule:satisfiable_rule2(r,p) -> pog(p), rules:rule(r).

total_achieved2[r,p]=t -> pog(p), rules:rule(r), int[32](t).
total_achieved2[r,p]+=picked[i,p]*potential[i,r,p].

rules:rule:satisfiable_rule2(r,p) -> total_achieved2[r,p]>=rules:rule:quantity[r].
rules:rule:satisfiable_rule(r,p) -> total_achieved2[r,p]<=rules:rule:quantity[r].
```

3.14 Switched Flow System (suiteSwitchedFlow)

This section presents an application of the BloxOptimize library to the switched flow system described in [AgarwalPhD]. The system, represented graphically below, consists of a tank being filled by two processes, A and B, and continuously emptied at a constant rate:



We model the behavior of the system in time defined as a series of discrete events:

```
EVENT(i), EVENT:id(i:id) -> int[64](id).
+system:Predicate:isOrdered[predicate]=value <-
   system:Predicate:fullName(predicate:"EVENT"),
   value = true.
```

The following parameters specify the initial level of the material in the tank, tank's maximum capacity, the minimum material level required at all times, and the flow-out rate:

```
// initial tank level
tankInit[]=v -> float[64](v).

// maximum tank capacity
tankMax[]=v -> float[64](v).

// minimum tank level
tankMin[]=v -> float[64](v).

// tank flow-out rate
tankOut[]=v -> float[64](v).
```

Process A represents a pump with two possible states, ON and OFF. When ON, the pump provides the tank with material, and incurs the operating cost, at each time event. There is a maximum length of time the pump can be continuously ON, after which it must be switched off. There is no operating cost when the pump is OFF, and there is a minimum length of time it must remain OFF before being turned on again. When the pump is turned on, a start-up cost is incurred. The following parameters specify the properties of process A:

```
// cost of operating process A at a single time event
opCostA[]=v -> float[64](v).

// cost of starting process A up
startUpCostA[]=v -> float[64](v).

// maximum number of time events that process A can be continuously ON
maxOnA[]=v -> int[64](v), v>=0, v<=100.

// minimum number of time events that must pass before process A is restarted
minOffA[]=v -> int[64](v), v>=0, v<=10.
```

```
// tank fill rate for process A
tankFillA[]=v -> float[64](v).
```

Process B represents a pump with two possible states, HIGH and LOW. At both settings the pump provides the tank with material, and incurs the operating cost, at each time event. After the pump is set to LOW, it must remain in this setting for a minimum length of time, and there is a start-up cost incurred when the pump is switched to HIGH again. The following parameters specify the properties of process B:

```
// cost of operating process B at LOW setting at a single time event
opCostLowB[]=v -> float[64](v).

// cost of operating process B at HIGH setting at a single time event
opCostHighB[]=v -> float[64](v).

// cost of switching process B from LOW to HIGH setting
startUpCostB[]=v -> float[64](v).

// minimum number of time events that must pass before process B
// can be switched to HIGH setting
minLowB[]=v -> int[64](v), v>=0, v<=10.

// tank fill rate for process B at LOW setting
tankFillLowB[]=v -> float[64](v).

// tank fill rate for process B at HIGH setting
tankFillHighB[]=v -> float[64](v).
```

We define problem variables to monitor, at every time event, the material level of the tank, and the state and clock updates of each process:

```
// Tank
tankLevel[i]=v -> EVENT(i), float[64](v), v>=0.
lang:solver:variable(`tankLevel).

// Process A
onA[i]=v -> EVENT(i), int[64](v), v>=0, v<=1.
offA[i]=v -> EVENT(i), int[64](v), v>=0, v<=1.
startUpA[i]=v -> EVENT(i), int[64](v), v>=0, v<=1.
shutOffA[i]=v -> EVENT(i), int[64](v), v>=0, v<=1.
clockA[i]=v -> EVENT(i), int[64](v), v>=0, v<=500.
clockJumpA[i]=v -> EVENT(i), int[64](v), v>=-500, v<=500.
lang:solver:variable(`onA).
lang:solver:variable(`offA).
lang:solver:variable(`startUpA).
lang:solver:variable(`shutOffA).
lang:solver:variable(`clockA).
lang:solver:variable(`clockJumpA).

// Process B
highB[i]=v -> EVENT(i), int[64](v), v>=0, v<=1.
lowB[i]=v -> EVENT(i), int[64](v), v>=0, v<=1.
startUpB[i]=v -> EVENT(i), int[64](v), v>=0, v<=1.
shutOffB[i]=v -> EVENT(i), int[64](v), v>=0, v<=1.
clockB[i]=v -> EVENT(i), int[64](v), v>=0, v<=500.
clockJumpB[i]=v -> EVENT(i), int[64](v), v>=-500, v<=500.
lang:solver:variable(`highB).
lang:solver:variable(`lowB).
```

```
lang:solver:variable(`startUpB).
lang:solver:variable(`shutOffB).
lang:solver:variable(`clockB).
lang:solver:variable(`clockJumpB).
```

We define domain filters to refer to all-but-first and all-but-last subsets of all time events:

```
// event that occurs before last possible event
BeforeLast(i) -> EVENT(i).
BeforeLast(i) <- EVENT:id(i:id), EVENT:last[]=j, EVENT:id(j:jd), id<jd.

// event that occurs after first possible event
AfterFirst(j) -> EVENT(j).
AfterFirst(j) <- EVENT:id(j:jd), EVENT:first[]=i, EVENT:id(i:id), jd>id.
```

We use constraints to correctly adjust the tank level:

```
// initial setting
EVENT:first[]=j -> tankLevel[j]>=tankInit[], tankLevel[j]<=tankInit[].
// tank level at time event j
EVENT:next[i]=j,
tankLevel[i]+tankFillA[]*onA[j]+tankFillHighB[]*highB[j]+tankFillLowB[]*lowB[j]-tankOut[]=v
->
tankLevel[j]>=v, tankLevel[j]<=v.

// tank level must be between allowed minimum and maximum at all times
EVENT(i) -> tankLevel[i]>=tankMin[].
EVENT(i) -> tankLevel[i]<=tankMax[].
```

to enforce correct state transitions of each process:

```
// process A takes (is in) exactly one action (state) at each time event
EVENT(i), startUpA[i]+shutOffA[i]+onA[i]+offA[i]=actionsA ->
      actionsA>=1, actionsA<=1.

//// Invalid actions/state transitions
// process A cannot go directly from OFF to ON
EVENT:next[i]=j -> offA[i]+onA[j]<=1.

// process A cannot be shut off when OFF
EVENT:next[i]=j -> offA[i]+shutOffA[j]<=1.

// process A cannot go directly from ON to OFF
EVENT:next[i]=j -> onA[i]+offA[j]<=1.

// process A cannot be started up when ON
EVENT:next[i]=j -> onA[i]+startUpA[j]<=1.

// process A cannot be ON after shut off
EVENT:next[i]=j -> shutOffA[i]+onA[j]<=1.

// process A cannot be started up immediately after shut off
EVENT:next[i]=j -> shutOffA[i]+startUpA[j]<=1.

// process A cannot be shut off in subsequent time events
EVENT:next[i]=j -> shutOffA[i]+shutOffA[j]<=1.
```

```
// process A cannot be OFF after start up
EVENT:next[i]=j -> startUpA[i]+offA[j]<=1.

// process A cannot be shut off immediately after start up
EVENT:next[i]=j -> startUpA[i]+shutOffA[j]<=1.

// process A cannot be started up in subsequent time events
EVENT:next[i]=j -> startUpA[i]+startUpA[j]<=1.



// process B takes (is in) exactly one action (state) at each time event
EVENT(i), startUpB[i]+shutOffB[i]+highB[i]+lowB[i]=actionsB ->
      actionsB>=1, actionsB<=1.

//// Invalid actions/state transitions
// process B cannot go directly from LOW to HIGH
EVENT:next[i]=j -> lowB[i]+highB[j]<=1.

// process B cannot go directly from HIGH to LOW
EVENT:next[i]=j -> highB[i]+lowB[j]<=1.

// process B cannot be shut off when on LOW
EVENT:next[i]=j -> lowB[i]+shutOffB[j]<=1.

// process B cannot be started up when on HIGH
EVENT:next[i]=j -> highB[i]+startUpB[j]<=1.

// process B cannot be on HIGH after shut off
EVENT:next[i]=j -> shutOffB[i]+highB[j]<=1.

// process B cannot be started up immediately after shut off
EVENT:next[i]=j -> shutOffB[i]+startUpB[j]<=1.

// process B cannot be shut off in subsequent time events
EVENT:next[i]=j -> shutOffB[i]+shutOffB[j]<=1.

// process B cannot be on LOW after start up
EVENT:next[i]=j -> startUpB[i]+lowB[j]<=1.

// process B cannot be shut off immediately after start up
EVENT:next[i]=j -> startUpB[i]+shutOffB[j]<=1.

// process B cannot be started up in subsequent time events
EVENT:next[i]=j -> startUpB[i]+startUpB[j]<=1.
```

and to ensure accurate clock updates:

```
// when process A is ON or OFF, process clock is ticking
AfterFirst(j) -> advanceClockA(j).
lang:isEntity[`advanceClockA]=false.
advanceClockA(j) -> EVENT(j).
advanceClockA(j) <- onA[j]+offA[j]+clockJumpA[j]=v, v>=2, v<=2.
advanceClockA(j) <- onA[j]+offA[j]=v, v>=0, v<=0.

EVENT:next[i]=j, clockA[i]+clockJumpA[j]=v -> clockA[j]>=v, clockA[j]<=v.

// at start up and shut off process clock is reset
```

```
EVENT(i) -> resetClockA(i).
lang:isEntity[`resetClockA]=false.
resetClockA(i) -> EVENT(i).
resetClockA(i) <- clockA[i]<=0.
resetClockA(i) <- startUpA[i]+shutOffA[i]<=0.

// process A can be continuously ON for no longer than maxOnA[] time events
EVENT(i) -> timedOnA(i).
lang:isEntity[`timedOnA]=false.
timedOnA(i) -> EVENT(i).
timedOnA(i) <- clockA[i]+onA[i]<=maxOnA[]+1.
timedOnA(i) <- clockA[i]+onA[i]<=0.

// process A can be restarted no sooner than after minOffA[] time events
EVENT:next[i]=j -> timedOffA(i,j).
timedOffA(i,j) -> EVENT(i), EVENT(j).
timedOffA(i,j) <- clockA[i]+startUpA[j]>=minOffA[].
timedOffA(i,j) <- startUpA[j]<=0, EVENT(i).

/*
 * This works
EVENT(j) -> timedOffA(j).
lang:isEntity[`timedOffA]=false.
timedOffA(j) <- EVENT:next[i]=j, clockA[i]+startUpA[j]>=minOffA[].
timedOffA(j) <- startUpA[j]<=0.
*/



// when process B is on LOW or HIGH setting, process clock is ticking
AfterFirst(j) -> advanceClockB(j).
lang:isEntity[`advanceClockB]=false.
advanceClockB(j) -> EVENT(j).
advanceClockB(j) <- highB[j]+lowB[j]+clockJumpB[j]=v, v>=2, v<=2.
advanceClockB(j) <- highB[j]+lowB[j]=v, v>=0, v<=0.

EVENT:next[i]=j, clockB[i]+clockJumpB[j]=v -> clockB[j]>=v, clockB[j]<=v.

// at start up and shut off process clock is reset
EVENT(i) -> resetClockB(i).
lang:isEntity[`resetClockB]=false.
resetClockB(i) -> EVENT(i).
resetClockB(i) <- clockB[i]<=0.
resetClockB(i) <- startUpB[i]+shutOffB[i]<=0.

// process B can be re-set to HIGH no sooner than after minLowB[] time events
EVENT:next[i]=j -> timedLowB(i,j).
timedLowB(i,j) -> EVENT(i), EVENT(j).
timedLowB(i,j) <- clockB[i]+startUpB[j]>=minLowB[].
timedLowB(i,j) <- startUpB[j]<=0, EVENT(i).
```

The problem objective is to minimize the total operational cost of the system:

```
lang:solver:minimal(`totalCost).
totalCost[]=v -> float[64](v).
totalCost[]=totalCostA[]+totalCostB[].

// Process A
```

```
totalCostA[]=v -> float[64](v).
totalCostA[]=totalOpCostA[]+totalStartUpCostA[].

totalOpCostA[]=v -> float[64](v).
totalOpCostA[]+=onA[_]*opCostA[].

totalStartUpCostA[]=v -> float[64](v).
totalStartUpCostA[]+=startUpA[_]*startUpCostA[].

// Process B
totalCostB[]=v -> float[64](v).
totalCostB[]=totalOpCostB[]+totalStartUpCostB[].

totalOpCostB[]=v -> float[64](v).
totalOpCostB[]+=highB[_]*opCostHighB[]+lowB[_]*opCostLowB[].

totalStartUpCostB[]=v -> float[64](v).
totalStartUpCostB[]+=startUpB[_]*startUpCostB[].
```

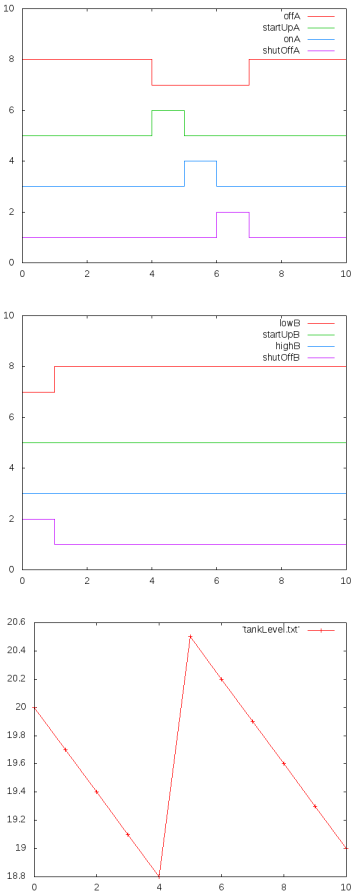
The problem specification involves several disjunctive constraints, and thus solving it requires the use of one of the transformations. Our specification works with both implemented transformations, however, because of the problem size, the COIN solver evaluation time is considerably long except for very small values of the systems parameters.

We now demonstrate how the behavior of the system depends on the values of the parameters. We start with the value set:

```
+EVENT(i), +EVENT:id(i:id) <- int64:range(0,10,1,id).

+tankInit[]=20.
+tankMax[]=25.
+tankMin[]=18.
+tankOut[]=1.8.

+opCostA[]=10.
+startUpCostA[]=50.
+maxOnA[]=3.
+minOffA[]=2.
+tankFillA[]=2.

+opCostLowB[]=2.
+opCostHighB[]=15.
+startUpCostB[]=4.
+minLowB[]=3.
+tankFillLowB[]=1.5.
+tankFillHighB[]=4.
```

The following graphs illustrate the corresponding behavior of the processes A and B, and the tank level (right). In each process graph, the topmost line represents the process' OFF/LOW setting, the second-from-top line indicates the start-up action, the third-from-top line represents the process' OFF/LOW setting, and the bottommost line indicates the shut-off action.

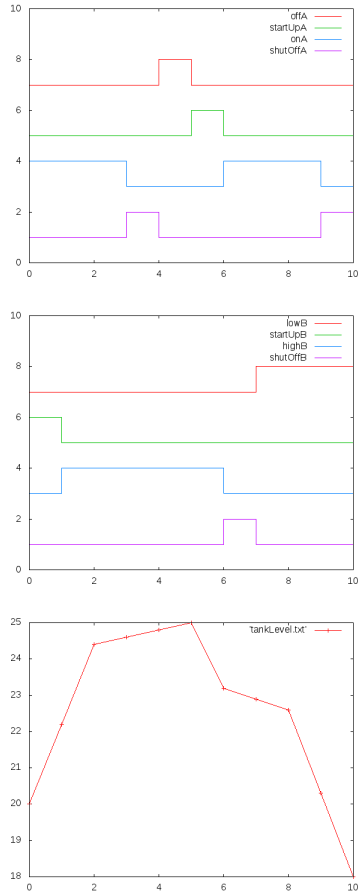
In the first graph we see that process A is in the OFF setting until time event 4, at which point the tank level (right graph) drops down to 18.8. Process A is then started for one time event, causing the tank level to raise to 20.5, and then shut off again at time event 6. Process B, with the HIGH setting operation cost of 15, is switched to LOW at time event 1, and remains in this state throughout the simulation:



Now, let's suppose that we increase the tank's flow-out rate from 1.8 to 3.8:

```
+EVENT(i), +EVENT:id(i:id) <- int64:range(0,10,1,id).

+tankInit[]=20.
+tankMax[]=25.
+tankMin[]=18.
+tankOut[]=3.8.

+opCostA[]=10.
+startUpCostA[]=50.
+maxOnA[]=3.
+minOffA[]=2.
+tankFillA[]=2.

+opCostLowB[]=2.
+opCostHighB[]=15.
+startUpCostB[]=4.
+minLowB[]=3.
+tankFillLowB[]=1.5.
+tankFillHighB[]=4.
```

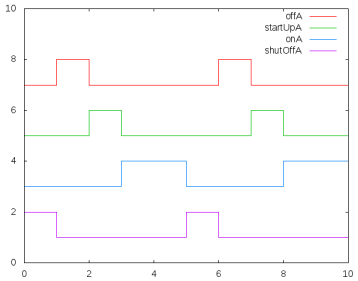
After this change the required material level in the tank can no longer be maintained with process B running on LOW setting. Thus, process B is switched to HIGH at time event 1, and operates at this setting until the tank is full. At that point (time event 6) process B is switched back to LOW. Process A helps fill up the tank in the initial phase of system lifetime, and is later started up to slow down tank material flow out:

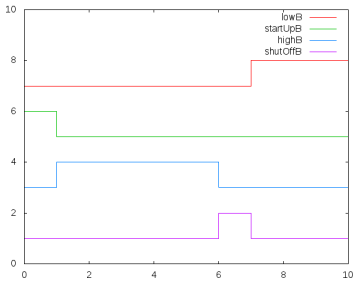


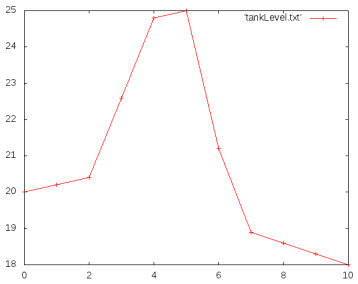
Next, with the increased tank flow-out rate, we reduce the start-up cost of process A:

```
+EVENT(i), +EVENT:id(i:id) <- int64:range(0,10,1,id).

+tankInit[]=20.
+tankMax[]=25.
+tankMin[]=18.
+tankOut[]=3.8.

+opCostA[]=10.
+startUpCostA[]=5.
+maxOnA[]=3.
+minOffA[]=2.
+tankFillA[]=2.

+opCostLowB[]=2.
+opCostHighB[]=15.
+startUpCostB[]=4.
+minLowB[]=3.
+tankFillLowB[]=1.5.
+tankFillHighB[]=4.
```

As a result, process A is started twice during the lifetime of the system:



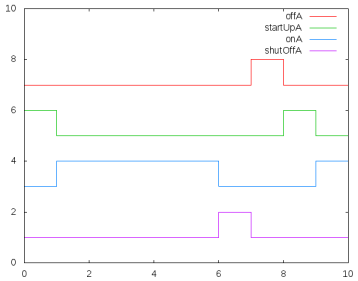


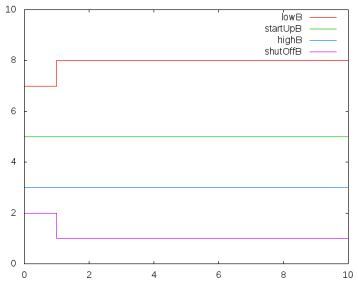


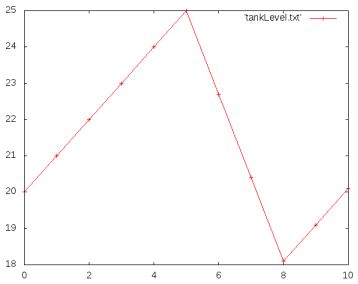
Finally, we extend the maximum time process A can be ON and increase its tank fill rate:

```
+EVENT(i), +EVENT:id(i:id) <- int64:range(0,10,1,id).

+tankInit[]=20.
+tankMax[]=25.
+tankMin[]=18.
+tankOut[]=3.8.

+opCostA[]=10.
+startUpCostA[]=5.
+maxOnA[]=5.
+minOffA[]=2.
+tankFillA[]=3.3.

+opCostLowB[]=2.
+opCostHighB[]=15.
+startUpCostB[]=4.
+minLowB[]=3.
+tankFillLowB[]=1.5.
+tankFillHighB[]=4.
```

This eliminates the need to operate the more costly process B on HIGH, again leaving control of the tank material level entirely to process A:


offA
startUpA
onA
shutOffA


lowB
startUpB
highB
shutOffB


"tankLevel.txt"

CHAPTER

FOUR

ISSUES

4.1 Constants on the left-hand side

One of the current limitations of BloxOptimize is that constants and variables have to be placed by the programmer on separate sides of a comparison.

For instance replacing:

```
x[s]=v1,y[s]=v2 -> v1+v2>=1.
```

with:

```
x[s]=v1,y[s]=v2 -> v1+v2-1>=0.
```

in the program from Section *Including Constraints* results in a run-time evaluation error.

4.2 Constraints filtered by numerical comparisons

The current version of BloxOptimize fails to generate a correct problem if in *Diet Problem (suiteDiet)* the constraint:

```
totalNutrAmt[n]=v -> NUTR(n), float[64](v).
totalNutrAmt[n]+=Buy[f]*amt[n,f].
NUTR(n),totalNutrAmt[n]=v1,nutrLow[n]=v2 -> v1>=v2.
```

is replaced by the following one, which enforces the constraint only in certain cases:

```
totalNutrAmt[n]=v -> NUTR(n), float[64](v).
totalNutrAmt[n]+=Buy[f]*amt[n,f].
NUTR(n),totalNutrAmt[n]=v1,nutrLow[n]=v2,v2>1 -> v1>=v2.
```

4.3 Variables in arithmetic expressions in the body of constraints

The current version of BloxOptimize fails to generate a correct program if in *Production-Transportation Problem (suiteProductionTransformation)* the constraint:

```
totalShipOut[o,p]=v -> ORIG(o), PRODUCT(p), float[64](v).
totalShipOut[o,p]+=Trans[o,_,p].
ORIG(o),PRODUCT(p),totalShipOut[o,p]=v1,Make[o,p]=v2,v2-v1=x-> x=0.
```

is replaced by the following one, which is logically equivalent but performs an arithmetic operation on the variables in the constraint body:

```
totalShipOut[o,p]=v -> ORIG(o), PROD(p), float[64](v).
totalShipOut[o,p]+=Trans[o,d,p].
ORIG(o),PROD(p),totalShipOut[o,p]=v1,Make[o,p]=v2 -> v2-v1=0.
```

4.4 Calculated parameters with disjunctions

The current version of BloxOptimize fails to generate a correct program if in *Indexed Variables* the EDB predicate *param* is replaced with an IDB predicate, which can be done by commenting out the initialization of *param* in the initialization block *test.init*:

```
+index(i),+index:id[i]=0.
+index(i),+index:id[i]=1.
//+param[i]=1 <- +index:id[i]=0.
//+param[i]=2 <- +index:id[i]=1.
+x_expected[i]=1 <- +index:id[i]=0.
+x_expected[i]=0 <- +index:id[i]=1.
```

and replacing it with two rules in the program block *suite.program*:

```
param[i]=n -> index(i),int[32](n).
param[i]=1 <- index:id[i]=0.
param[i]=2 <- index:id[i]=1.
```

4.5 Key-less variable predicates

The current version of BloxOptimize fails to generate a correct program if a variable predicate is defined without a key argument of entity type. For example, the following encoding of the program from the *The Simplest Problem* Section is incorrect:

```
x[]=v -> int[32](v), v<=3, v>=0.
lang:solver:variable(`x).
```

CHAPTER

FIVE

USAGE NOTES

5.1 Optimization library run-time directive

BloxOptimize is linked as an external library with the `-lib` option given at run time to the testing program (e.g. `bloxunit`). For instance, to run our simplest test suite discussed in Section *The Simplest Problem* from the test suite directory `LogicBlox/BloxUnit/testsuites/applications-optimization/datalog-existential`, one should say:

```
'bloxunit -lib BloxOptimize -suite suiteOneVar'
```

Alternatively, the `-lib BloxOptimize` directive can be provided in the *suite.options* file in the suite directory. At this time, however, the directive supplied in this way might be not recognized properly by the engine if the file also contains some other options.

5.2 Calling solvers using external server

BloxOptimize has been enhanced with support for client/server architecture, in which the process evaluating a DatalogLB program calls a remote server to handle the communication with an external solver. The architecture is implemented using Google protocol buffers to serialize the data passed between the client and the server processes, and Boost asio to model their communication via tcp/ip. The client/server architecture is available under both Linux and Windows.

The client/server execution is enabled by setting the environment variable `USE_BLOXOPTIMIZE_SERVER`:

```
'export USE_BLOXOPTIMIZE_SERVER='
```

The value of the variable can be set to identify the host and port to be used by the server, for example:

```
'export USE_BLOXOPTIMIZE_SERVER=localhost:8001'
```

sets the host name to localhost and the port to 8001.

The server is defined in the new project LogicBlox/BloxOptimizeServer, which generates the executable `bloxoptimizeserver`. The executable can be invoked with the following options:

- `bloxoptimizeserver -p [--port] arg` starts the server that listens for connections at the port `arg` (by default `arg` is set to `8001`);
- `bloxoptimizeserver --shutdown` terminates the server existing in the provided port;

- `bloxoptimizeserver --logLevel arg` specifies the logging level for the server; possible values of `arg` are: `all`, `debugDetail`, `debug`, `information`, `warning`, `error`, `none` (by default `arg` is set to `debug`);
- `bloxoptimizeserver --logFile arg` specifies the name of the log file for the server process (by default `logFile` set to `bloxOptimizeServer.log`);
- `bloxoptimizeserver -?` summarizes the above options.

The server can be launched as a new process on the local machine. At this point, such a server process is never terminated by the system, but rather must be terminated manually by the programmer using the `bloxoptimizeserver --shutdown` or `kill -9` command.

5.3 Handling Disjunction

Existing automated MP solvers lack direct support for problems involving disjunction. In order to be accepted by a solver, such problems must be first converted into non-disjunctive specifications using a dedicated algebraic transformation method. To accommodate problems with disjunctive constraints in LogicBlox, we added necessary extensions to the language syntax, and implemented two transformations for eliminating disjunction, which we briefly descibe in the following sections:

5.3.1 Convex Hull Transformation

The convex hull transformation generates the tightest possible relaxation (the convex hull) of the original disjunctive problem. The resulting programs may be more efficient than those produced by the big-M transformation. However, as the convex hull transformation relies on numerous auxiliary variables and equations, for some programs its computational costs may offset the benefits.

Given a general form of a disjunctive constraint:

$[A^1x \leq b^1] \vee [A^2x \leq b^2]$

the convex hull transformation rewrtites it to the folowing conjunction of linear constraints:

$A^1\bar{x}^1 \leq b^1y_1 \quad \wedge \quad A^2\bar{x}^2 \leq b^2y_2 \quad \wedge \quad y_1 + y_2 = 1 \quad \wedge \quad x = \bar{x}^1 + \bar{x}^2$

where $y_i \in \{0, 1\}$, and $\bar{x}^i$ are fresh variables replacing in each disjunct the variables from the original constraint. The goal of introducing these new variables is to *disaggregate* the original disjuncts, that is, to make them have no variables in common.

Convex hull is the default disjunctive constraint transformation in LogicBlox. DatalogLB programmers may invoke it explicitly by including the program directive:

```
'optimization:disjunctionRewrite[]="convex hull".'
```

5.3.2 Big-M Transformation

The big-M transformation for disjunctive constraints uses expression bounds and boolean variables to enable individual evaluation of the original disjuncts.

Given a general form of a disjunctive constraint:

$[A^1x \leq b^1] \vee [A^2x \leq b^2]$

the big-M transformation rewrtites it to the folowing conjunction of linear constraints:

$A^1x - b^1 \leq M^1(1 - y_1) \quad \wedge \quad A^2x - b^2 \leq M^2(1 - y_2) \quad \wedge \quad y_1 + y_2 = 1$

where $y_i \in \{0, 1\}$, and M^i (referred to as big-M parameters) are known upper bounds on $A^i x - b^i$. The selection of these parameters determines the efficiency of the transformation. Our implementation automatically computes the big-M values based on the interval arithmetic.

DatalogLB programmers may invoke the big-M transformation by including the program directive:

```
`optimization:disjunctionRewrite[]="big M".`
```

5.4 Interfacing the solvers

The support for mathematical programming provided by the BloxOptimize library is based on the library's interface to external MP solvers. At this time BloxOptimize supports the following platforms:

- **Optimization Services (OS)**—an open-source project, developed as part of the Computational Infrastructure for Operations Research (COIN-OR) initiative, and aimed at providing a framework of standardized interfaces to a number of free and commercial MP environments,
- **Gurobi**—a commercial, simplex-based solver for linear and mixed-integer programming problems,
- **lp-solve**—a free linear (integer) programming solver based on the revised simplex method and the Branch-and-bound method for the integers.

The third-party libraries for all interfaces are located in the LogicBlox development tools directory, referred to as `LB_DEVTOOLS`, and named `Linux-2.6-x86_64-gcc-4.1/` (under Linux) or `Windows-XP-i386-vc8/` (under Windows). After checking out/updating the `LB_DEVTOOLS` directory, follow the instructions for building/using the specific interface in the corresponding section below.

5.4.1 COIN-OS

BloxOptimize currently supports both 1.1 and 2.0 versions of COIN-OS under Linux, and the 1.1 version under Windows.

To use COIN-OS 1.1 on either platform:

- extend `LD_LIBRARY_PATH` with: `$LB_DEVTOOLS/COIN_OS/lib`;
- build BloxOptimize.

To use COIN-OS 2.0 under Linux:

- extend `LD_LIBRARY_PATH` with: `$LB_DEVTOOLS/COIN_OS_2/lib`;
- since COIN-OS 2.0 works only in the client/server architecture, set the `USE_BLOXOPTIMIZE_SERVER` variable as described in Section *Calling solvers using external server*;
- in the `$LB_DEVTOOLS/COIN_OS_2/` directory execute the `install.sh` script to create the symbolic links to the library files;
- in the `$LB_DEVTOOLS/COIN_OS_2/` directory create a (possibly empty) file named `coin2.ok`;
- build BloxOptimize;
- build BloxOptimizeServer.

The interface to COIN-OS is the default interface of BloxOptimize. COIN-OS enables access to several solvers, and selects the solver to be used for given program based on the program data. DatalogLB programmers may choose specific solver (e.g., *Cbc* below) to be used by including in their program the line:

```
`optimization:solverName[]="cbc".`
```

5.4.2 Gurobi

BloxOptimize currently supports the 2.0 version of Gurobi under Linux. The solver is enabled by including in the DatalogLB program the line:

```
`optimization:solverName[]="gurobi".`
```

5.4.3 lp-solve

BloxOptimize currently supports the 5.5.0.15 version of lp-solve under Linux and Windows. To use the solver on either platform, extend `LD_LIBRARY_PATH` with `$LB_DEVTOOLS/LP_SOLVER`. The solver is then enabled by including in the DatalogLB program the line:

```
`optimization:solverName[]="lp_solve".`
```

CHAPTER

SIX

TUTORIAL: TASK SCHEDULING

6.1 The Problem

In this tutorial we apply BloxOptimize library to solving the family of task scheduling problems generally specified as follows:

- There exists a set of tasks which need to be assigned to people. Each task has a skill level that is required to complete it. Each task also has a deadline. Tasks may depend on each other, and if task `t2` depends on task `t1`, then `t2` cannot be scheduled to start before `t1` has been completed;
- There exists a set of people, of whom each has as a skill level;
- There exists a set of consecutive time periods/days;
- The goal is to find a schedule which assigns a person to a task on a given day, and at the same time minimizes certain specified cost.

The cost of the assignemnt is computed independently for each scheduling period, by means of the predicate `cost[person,task,day]`. The value of the predicate for given key tuple `(p,t,d)` depends on several factors, including the qualifications of the person `p`, the skill level necessary for completing the task `t`, and the type of the time period `d`: It is more expensive to assign overqualified people to a task that requires a skill level lower than they have, and assigning people to work during the weekend costs more than on regular week days.

6.2 The Implementation

We start with the simple version of the problem in the following model.

First, we declare the `TASK`, `PERSON`, and `TIME` entities and their properties:

```
1  TASK(t), TASK:name(t:n) -> string(n).
2  TASK:duration[t]=n -> TASK(t), int[32](n).
3  TASK:requiresLevel[t]=n -> TASK(t), int[32](n).
4  lang:ordered(`TASK).
5
6  //task t1 depends on task t2
7  TASK:depends(t1,t2) -> TASK(t1), TASK(t2).
8  TASK:depends:trans(t1,t2) <- TASK:depends(t1,t2).
9  TASK:depends:trans(t1,t2) <- TASK:depends:trans(t1,t3),
10                              TASK:depends:trans(t3,t2).
11
12 TASK:deadline(t,d) -> TASK(t), TIME(d).
13
```

```
14
15 TIME(t), TIME:id(t:id) -> int[32](id).
16 lang:ordered(`TIME).
17 TIME:before(t1,t2) <- TIME(t1), TIME(t2), TIME:id[t1] <= TIME:id[t2].
18 TIME:before:neq(t1,t2) <- TIME(t1), TIME(t2), TIME:id[t1] < TIME:id[t2].
19 TIME:after(t1,t2) <- TIME(t1), TIME(t2), TIME:id[t1] > TIME:id[t2].
20
21 PERSON(t), PERSON:name(t:n) -> string(n).
22 PERSON:skill[p]=s -> PERSON(p), int[32](s).
23 lang:ordered(`PERSON).
24
25 qualifiedFor(p,t) <- PERSON:skill[p] >= TASK:requiresLevel[t].
26 notQualifiedFor(p,t) <- PERSON(p), TASK(t), !qualifiedFor(p,t).
27
28 overtime[]=v -> float[64](v).
```

The entities are used to define the `cost` predicate. In this model the cost is computed for each assignment point, and is dictated by the difficulty level of the task and the skill level of the person assigned to it. A small cost penalty is added when tasks last longer, are done with more skill, and at a later date.

```
1  cost[t,p,d]=v -> TASK(t), PERSON(p), TIME(d), float[64](v).
2  cost[t,p,d]=v <- TASK:requiresLevel[t]=level,
3                TASK:duration[t]=duration,
4                PERSON:skill[p]=skill,
5                TIME(d),
6                TIME:id[d]=day,
7                v = (level + (level-skill)*0.2) + float64:log[int32:float64:convert[duration*skill*day]]
8                skill>=level,
9                int32:mod[day,7]>1.
10 cost[t,p,d]=v <- TASK:requiresLevel[t]=level,
11               TASK:duration[t]=duration,
12               PERSON:skill[p]=skill,
13               TIME(d),
14               TIME:id[d]=day,
15               v1 = (level + (skill-level)*0.2) + float64:log[int32:float64:convert[duration*skill*day]
16               v = v1*overtime[],
17               skill>=level,
18               int32:mod[day,7]<=1.
```

The main part of our model is the specification of the variable `SCHEDULE` and the constraints on it. The variable is declared as follows:

```
1 SCHEDULE[task,person,time]=v -> TASK(task),
2                                 PERSON(person),
3                                 TIME(time),
4                                 int[32](v),
5                                 v >= 0, v <= 1.
6 lang:solver:variable(`SCHEDULE).
```

The constraints on the `SCHEDULE` variable embody a particular set of business rules concerning the scheduling tasks. In our discussion we also speculate how these constraints can be extended to explore the design space of possible rules.

The first constraint ensures that the total amount of work spent on a particular task matches the task's length:

```
TASK(t), TASK:duration[t]=d, scheduledTask[t]=v -> v >= d.
scheduledTask[task]=v -> TASK(task), int[32](v).
scheduledTask[task]=v <- agg<<v=total(z)>> SCHEDULE[task,_,_]=z.
```

The second constraint prevents assignments where the required skill level of a task is higher than the the skill level of a person assigned to it. This is ensured by setting `SCHEDULE[t,p,d]` to `0` for every person `p` not qualified sufficiently for the task `t`. An interesting variation of this requirement might be represented by combining this with the previous constraint to state that a task at the skill level `l1` may be assigned to a person of skill level `l2`, `l2<=l1`, at the cost of increasing the length of the task:

```
notQualifiedFor(person,task), TIME(d), SCHEDULE[task,person,d]=v -> v <= 0.
```

The next constraint requires that a person is assigned at most one task on any given day:

```
PERSON(p), TIME(d), scheduledPersonDay[p,d]=v -> v <= 1.
scheduledPersonDay[person,d]=v -> PERSON(person), TIME(d), int[32](v).
scheduledPersonDay[person,d]=v <- agg<<v=total(z)>> SCHEDULE[_,person,d]=z.
```

A natural extension of the above constraint, in which the constant `1` is replaced with the parameter `parallelization[p]`, determines how many tasks are assigned at the same time individually for each person:

```
PERSON(p), TIME(d), scheduledPersonDay[p,d]=v -> v <= parallelization[p].
scheduledPersonDay[p,d]=v -> PERSON(p), TIME(d), int[32](v).
scheduledPersonDay[p,d]=v <- agg<v=total(z)> SCHEDULE[_,p,d]=z.
```

A more realistic approach would be to capture the difficulty of tasks with the amount of parallelization-per-task-per-person to prevent overloading a person with too many difficult tasks. This can be represented relatively easily in the current scheme.

The next constraint ensures that only one person works on a task on any given day:

```
scheduledTaskDay[task,d]=v -> TASK(task), TIME(d), int[32](v).
scheduledTaskDay[task,d]=v <- agg<<v=total(z)>> SCHEDULE[task,_,d]=z.
TASK(t), TIME(d), scheduledTaskDay[t,d]=v -> v<=1.
```

A similar constraint, with the parameter `max_people[t]` replacing the constant `1`, could be used to ensure that each task is assigned no more a certain number of people:

```
scheduledTaskDay[task,d]=v -> TASK(task), TIME(d), int[32](v).
scheduledTaskDay[task,d]=v <- agg<<v=total(z)>> SCHEDULE[task,_,d]=z.
TASK(t), TIME(d), scheduledTaskDay[t,d]=v -> v<=max_people[t].
```

The next constraint ensures that no task is scheduled after its deadline:

```
TASK(t), TASK:deadline(t,deadline), TIME:after(d,deadline), PERSON(p), SCHEDULE[t,p,d]=v -> v=0.
```

An interesting, and perhaps more realistic, alternative constraint would be to allow some slippage of a task completion time, and then to considerably increase the cost for each day past the deadline.

The next constraint, formulated with the indispensable help from our dear colleague Monsieur Oget, illustrates a relatively common pattern when expressing this sort of rule. We introduce a binary auxiliary variable `SLACK` with the domain `TASK*PERSON`, the role of which is to select the people who have been selected for the particular task (line 3). The business requirement is enforced by the next constraint (lines 4-5), which ensures that the sum of all people selected for any task is `1`. Thus, because of the sum constraint, only one person can be selected. Note that the number of people allowed to work on a given task can be changed simply by adjusting the numeric value in the constraint formula.

```
SLACK[task,person]=v -> TASK(task), PERSON(person), int[32](v), v>=0, v<=1.
lang:solver:variable(`SLACK).
TIME(d), TASK(t), PERSON(p), SLACK[t,p]=v1, SCHEDULE[t,p,d]=v2 -> v1>=v2.
sumPeopleSlack[t]=v <- agg<<v=total(z)>> SLACK[t,_]=z.
TASK(t), sumPeopleSlack[t]=v -> v=1.
```

In the last constraint, another auxiliary variable, `WORKDONE`, is introduced to enforce the inter-task dependencies. For each task `t`, WORKDONE[t,d] records the total number of days spent working on the task `t` up to the day `d`. The constraint ensures that, for each pair of tasks `t1` and `t2` such that `t2` depends on `t1`, the task `t2` is scheduled on a day `d` only if there are at least `duration[t1]` days spent on task `t1` before `d`. If the number of days assigned to `t1``prior to ``d``is smaller than ``duration[t1]`, `SCHEDULE[t2,p,d]` is set to `0` for all persons `p`:

```
WORKDONE[t,d]=v -> TASK(t), TIME(d), int[32](v).
lang:solver:variable(`WORKDONE).
TASK(t), TIME(d), WORKDONE[t,d]=v1, workUpTo[t,d]=v2 -> v1=v2.
workUpTo[t,d]=v <- agg<<v=total(z)>> z=SCHEDULE[t,_,d1], TIME:before:neq(d1,d).

TASK:depends(t1,t2), PERSON(p), TIME(d), WORKDONE[t2,d]=v1, SCHEDULE[t1,p,d]*TASK:duration[t2]=v2 ->
```

By extending the above formula with parameters we can refine the constraint e.g. to allow scheduling the task `t2`, depending on the task `t1`, after the completion of some percentage, rather than all of, `t1`.

The objective function simply computes the total cost of all task assignments:

```
objective[]=v -> float[64](v).
objective[]=v <- agg<<v=total(z)>> z=int32:float64:convert[SCHEDULE[t,p,d]]*cost[t,p,d].
lang:solver:minimal(`objective).
```

A slight elaboration

Suppose now that the task assignment solution that we would like to compute should not only minimize the assignment cost, but also satisfy certain preferences such as assigning a particular person to a particular task, or not assigning a particular person to a particular task. One way to achieve this is to represent these preferences as additional program constrains. However, satisfying the preference constraints would likely cost us some optimality, as any schedule that satisfies the constraints may, and probably will, cost more. Instead, we use a technique of modeling a trade-off between preferences and constraints, which allows us to state that we are willing to accept sub-optimality with respect to cost (within some boundaries), and trade it for preferences. In the first step out our technique we solve the problem as before and obtain the optimal schedule minimizing the assignment cost. In the second step, we create another scheduling problem, with the variable `SCHEDULE'` subject to all the constraints on the variable `SCHEDULE` from the original problem, plus the constraint that ensures that the total cost of `SCHEDULE'` is no greater than the total amount of work on `SCHEDULE` plus some percentage. Finally, we change the objective function in the new program to reflect the preferences.

CHAPTER

SEVEN

REFERENCES

CHAPTER

EIGHT

GLOSSARY

constraint A constraint is a restriction on the values that the variables may take in an optimization problem solution.

expected value An expected value for a *variable* in an optimization problem is the value which the variable is expected to assume in the solution of that problem

objective function An objective function is a function that uses the information provided by the parameters to compute values for each *variable*.

parameter A parameter is an idenitifer representing the information supplied as a part of the specification of an optimization problem. Parameters are used to compute values for the variables.

variable A variable is an identifier for an unknown value in the specification of an optimization problem. The value of the variable is determined in the problem solution.

BIBLIOGRAPHY

[AgarwalPhD] Ashish Agarwal. *Logical Modeling Frameworks for the Optimization of Discrete-Continuous Systems*. PhD Thesis, Carnegie Melon University, 2006.

INDEX