Modelling

Christina Burt, Stephen J. Maher, Jakob Witzig

Zuse Institute Berlin
Berlin, Germany

29th September 2015
Modelling Languages

Jakob Witzig
Algebraic Modelling Languages are high-level computer programming languages. They

- provide a concise, readable way to express/formulate a mathematical problem,
- do not solve the problems directly, but rather, call external solvers,
- generally contain a mix of declarative and procedural elements.

Examples of modelling languages include Xpress-Mosel, ZIMPL, AIMMS, AMPL, GAMS and OPL.
The benefits of using a modelling language to implement a model include:

- can be a fast way to implement a model,
- easy to learn due to few basic language elements; short learning curve to achieve high expressivity,
- syntax is usually very similar to mathematical notation, making it intuitive,
- simplified expression of some elements of math programming models, such as sets and logical expressions.
Modelling Languages: Pros and Cons

ZIMPL:

```plaintext
set Warehouses := {1,2,3,4};
set Plants := {1,2};
var transport[Warehouses * Plants] integer;
```

Gurobi:

```plaintext
Warehouses = [1,2,3,4]
Plants = [1,2]
transport = []
for w in warehouses:
    transport.append([])
    for p in plants:
        transport[w].append(m.addVar(obj=transCosts[w][p],
                                      name="Trans%d.%d" % (p, w)))
```
The drawbacks of using a modelling language include:

- access to solver features, such as callbacks, is often difficult (or not possible)
- the solving is decoupled from the modelling process to some extent – the user still must understand how to model well for the target algorithm,
- the language interpreter may perform inefficient conversions compared to what a reformulation may achieve.
Today we will do a modelling exercise that will illustrate:

- how quickly you can learn a modelling language,
- how quickly you can implement a model and obtain a solution from an installed solver.
Some Modelling Languages

- **ZIMPL**: for writing down a model quickly and output a format for LP, MIP and MINLP that can be read into any LP/MIP solver (and SCIP for MINLP);

- **GAMS**: contains procedural elements for modifying the data and model during the procedure, can switch between solvers, has an IDE with tree-view and model debugging;

- **AIMMS**: contains procedural elements and tools for deployment, e.g. it is possible to write a GUI for a model-application for partners who do not want to see the model itself

- **AMPL**: integrates a modelling language, command language (for debugging and analysing), and a scripting language for modifying the data and model during the procedure.
Getting Started with ZIMPL

Jakob Witzig
What is ZIMPL?

- **Zuse Institute Mathematical Programming Language**
- Free software, first release 2001
What is ZIMPL?

- **Zuse Institute Mathematical Programming Language**
- Free software, first release 2001
- Language to translate mathematical models into (mixed) integer programs.
What is ZIMPL?

- **Zuse Institute Mathematical Programming Language**
- Free software, first release 2001
- Language to translate mathematical models into (mixed) integer programs.
- ZIMPL is part of the SCIP Optimization Suite ([http://scip.zib.de/](http://scip.zib.de/))
Each line ends with a semicolon
Each line ends with a semicolon

6 different statements:

- Sets,
- Parameters,
- Variables,
- Objective,
- Constraints,
- Function definition, and print commands.
ZIMPL Format & Functions

- Each line ends with a semicolon
- 6 different statements:
  - Sets,
  - Parameters,
  - Variables,
  - Objective,
  - Constraints,
  - Function definition, and print commands.
- Rational arithmetic functions (e.g., modulo, factorial, min/max of a set, rounding, . . . )
ZIMPL Format & Functions

- Each line ends with a semicolon
- 6 different statements:
  - Sets,
  - Parameters,
  - Variables,
  - Objective,
  - Constraints,
  - Function definition, and print commands.
- Rational arithmetic functions (e.g., modulo, factorial, min/max of a set, rounding, ...)
- Double precision functions: square root, logarithm, exponential function
ZIMPL Format & Functions

- Each line ends with a semicolon
- 6 different statements:
  - Sets,
  - Parameters,
  - Variables,
  - Objective,
  - Constraints,
  - Function definition, and print commands.

- Rational arithmetic functions (e.g., modulo, factorial, min/max of a set, rounding, ...)
- Double precision functions: square root, logarithm, exponential function
- Set related functions (e.g., cross product, union, argmin, ...)

Burt, Maher, Witzig – Modelling
ZIMPL Format & Functions

- Each line ends with a semicolon
- 6 different statements:
  - Sets,
  - Parameters,
  - Variables,
  - Objective,
  - Constraints,
  - Function definition, and print commands.
- Rational arithmetic functions (e.g., modulo, factorial, min/max of a set, rounding, . . .)
- Double precision functions: square root, logarithm, exponential function
- Set related functions (e.g., cross product, union, argmin, . . .)
- Input stream to read in data files
Executing ZIMPL

To get an LP file of example.zpl

\texttt{zimpl example.zpl}

Use SCIP to read in directly and solve

\texttt{scip -f example.zpl}
Set Definitions:

set A := \{1 .. 4\};
set B := \{1 to 3\};
set D[B] := \langle 1\rangle\{"a", "b"\}, \langle 2\rangle\{"c"\}, \langle 3\rangle\{"c", "e", "f"\};
set E := \{"monkey", "giraffe", "elephant"\};

Set Operations:

set C := A cross C;
set P[] := powerset(D);
set I := indexset(P);
set S[] := subsets(I, 2);
set U := union \langle i \rangle in I : D[i];
Parameters

- **Indexsets:**
  ```plaintext
  set I := {1 .. 10};
  set J := {"a", "b", "c", "x", "y", "z"};
  ```

- **Parameters:**
  ```plaintext
  param h[|I*J|] := |
  "a", "c", "x", "z" |
  | 1 | 12, 17, 99, 23 |
  | 3 | 4, 3, -17, 66*5.5 |
  | 5 | 2/3, -4, 3, abs(-4) |
  | 9 | 1, 2, 0, 3 | default -99;

  param g[|I*I*|] := |
  1, 2, 3 |
  | 1,3 | 0, 0, 1 |
  | 2,1 | 1, 0, 1 |
  |

  param k[|I*|] := <4,7> 89, <4,8> 67, <4,9> 55, <5,7> 12,
  <5,8> 13, <5,9> 14, <1,2> 17, <3,4> 99;
  ```
Reading Sets and Parameters

- **Input file** `data.txt`:
  
  1 2  ab  con1  
  2 3  bc  con2  
  4 5  de  con3  

- **Read Sets**:
  
  ```
  set Q := \{ \text{read "data.txt" as "]4s"} \};
  ```

- **Read Parameter**:
  
  ```
  param cost [Q] := \text{read "data.txt" as "2n 1n"};
  ```
Variables

- Types: binary, integer, and real

- Implicit bounds are $[0, \infty]$ 

- Declare binary/integer variables as implicit

- Example:

  ```
  var x1;
  var x2 binary;
  var x3 integer >= -infinity;
  var y[A] real >= 2 <= 18;
  var z<[a,b] in C] integer
    >= a*10 <= if b <= 3 then p[b] else infinity end;
  var w implicit binary;
  ```

Burt, Maher, Witzig – Modelling
Objective Functions

- Syntax: \texttt{sense name : term;}

- Can be \texttt{minimize} or \texttt{maximize}

- Must be at most one objective function

- Example:
  
  \begin{verbatim}
  minimize obj : 3*x1 - 1*x2;
  minimize cost : sum <i,j> in E with i < j : c[i,j] * x[i,j];
  maximize profit : sum <i> in I : p[i] * x[i];
  \end{verbatim}
Constraints

- Syntax: \texttt{subto name : term sense term;}

- Sense can be \texttt{\leq}, \texttt{==}, and \texttt{\geq}

  \texttt{subto c1: sum <i,j> in E with \ i < j \ and \ j \neq 1 : x[i,j] == 1;}

  \texttt{subto c2: for all <i> in I do if i < 5 \ then \ 3 \ y[i] \ else \ 2 \ y[i] end;}

  \texttt{subto c3: for all <i,j> in E do if i < j \ then \ sum <k> in K : y[k,i,j] == 1 \ and \ x[i,j] == 2 \ else \ sum <k> in K : y[k,i,j] == 0 end;}

Constraints

- Syntax: `subto name : term sense term;`

- Sense can be `<`, `==`, and `>`
  
  `subto c1 : sum <i,j> in E with i < j and j != 1 : x[i,j] == 1;`

- Can be combined with if-conditions
  
  `subto c2 : forall <i> in I do
  if i <= 5 then 3*y[i] >= 1
  else -2*y[i] <= 0 end;`
Constraints

- Syntax: `subto name : term sense term;

- Sense can be $\leq$, $==$, and $\geq$

  ```
  subto c1: sum <i, j> in E with i < j and j != 1 : x[i, j] == 1;
  ```

- Can be combined with if-conditions

  ```
  subto c2: for all <i> in I do
  if i <= 5 then 3*y[i] >= 1
  else -2*y[i] <= 0 end;
  ```

- Combining with and

  ```
  subto c3: for all <i, j> in E do
  if i < j then sum <k> in K : y[k, i, j] == 1 and x[i, j] == 2
  else sum <k> in K : y[k, i, j] == 0 end;
  ```
Functions Definition:

- Syntax: `def* name(input) := term;`
- `*` = `numb`, `strg`, `bool`, and `set`
- Example:

```
 defnumb dist(a,b) := sqrt(a*a + b*b);
defbool smal(a,b) := a < b;
defset bigg(i) := { <j> in K with j > i };
```
Functions Definition:

- Syntax: `def* name(input) := term;`
- `* = numb, strg, bool, and set`
- Example:

```
defnumb dist(a,b) := sqrt(a*a + b*b);
defbool smal(a,b) := a < b;
defset bigg(i) := { <j> in K with j > i };```

Print Commands:

- Print Sets:

```
set l := {1 .. 10};
do print l;
do print "Cardinality of l:", card(l);
```

- Print Supersets:

```
do forall <i> in l with i >= 4 do print sqrt(i);```
Exercise: ZIMPL

The SCIP Team
Exercise: ZIMPL

Implement the model of the equitable coach problem you have created before in ZIMPL.

Use the code skeleton: \texttt{/home/exercise/modelling/Exercise_ECP.zpl}

The ZIMPL user guide is located at: \texttt{/home/exercise/modelling/}
The Code Skeleton

- You can execute the *.zpl file with:

  zimpl Exercise_ECP.zpl -D FILE=(test_data)

- You have to . . .
  - . . . complete the reading of parameters and sets
  - . . . define a function
  - . . . define your own variables
  - . . . define the model itself (objective function, constraints, etc.)
Exercise: ZIMPL

Implement the model of the equitable coach problem you have created before in ZIMPL.

Use the code skeleton: /home/exercise/modelling/Exercise_ECP.zpl

The ZIMPL user guide is located at: /home/exercise/modelling/