

TAILOR Tutorial

Andrea Rendl

July 17, 2008

1 Introduction

This tutorial introduces the tool TAILOR that converts constraint problem models formulated in the solver-independent modelling language ESSENCE' to the input format of the Constraint Solver MINION.

ESSENCE' is a solver-independent modelling language for Constraint Programming. It provides means to define variables, constants, parameters and offers a great range of constraint expressions, including complex constructs, such as quantifications.

MINION is a fast scalable Constraint solver. However, modelling problems in MINION's input language is time-consuming and tedious because of its primitive structure (it can be compared to writing a complex program in machine language).

TAILOR converts ESSENCE' problem instances into MINION format and applies reformulations (such as common subexpression elimination) during this process to enhance the problem model.

1.1 Installing TAILOR

TAILOR comes together with MINION on <http://minion.sourceforge.net/> as an executable Java `jar` [1] file. It can also be downloaded as a standalone version from <http://www.cs.st-and.ac.uk/~andrea/tailor/tailor.tar.gz>.

1.2 Running TAILOR

The Java `jar` file `tailor.jar` can be executed either by double-clicking or in the command line with the command

```
java -jar tailor.jar
```

This initiates the graphical user interface of TAILOR. More details about the command line version are given with

```
java -jar tailor.jar -help
```

2 An Introduction to ESSENCE'

2.1 Types and Domains

Types and domains play a similar role; they prescribe a range of values that a variable can take. Types denote non-empty sets that contain all elements that have a similar structure, whereas domains denote possibly empty sets drawn from a single type. In this manner, each domain is associated with an underlying type. For example integer is the type underlying the domain comprising integers between 1 and 10.

ESSENCE' is a strongly typed language; every expression has a type, and the types of all expressions can be inferred and checked for correctness. Furthermore, ESSENCE' is a finite-domain language; every decision variable is associated with a finite domain of values.

The atomic types of ESSENCE' are `int` (integer), `bool` (Boolean) and user-defined enumerated types. There is also a compound type, array (or matrix), type that is constructed of atomic types.

There are three different types of domains in ESSENCE': boolean, integer and matrix/array domains. Boolean and integer domains are both atomic domains; array domains are built from atomic domains.

Decision variables and quantified variables need to be associated with a *finite* domain. The infinite integer domain, `int`, is only valid with parameters. Decision variables are not allowed as domain elements.

Boolean Domains `bool` is the Boolean domain consisting of *false* and *true*.

Integer Domains An integer domain is a range of integers that can be either continuous, e.g. `int(1..10)`, or sparse, e.g. `int(1,3,5)`. Continuous domains are considered to be the empty domain if the lower bound is greater than the upper bound, such as in `int(10..1)`. The elements of sparse domains need to be ordered, hence `int(1,5,3)` is not valid.

Array Domains An array is defined by the keyword `matrix`, followed by its dimension and the base domain (over which the variables range). For instance,

`NAME1 : matrix indexed by [int(1..10)] of int(1..5)`

stands for a 1-dimensional array of 10 elements where each element ranges from `int(1..5)`. The index domain states how to dereference arrays. In the example above, `NAME1[1]` dereferences the first element because the index domain starts with 1. However, consider `NAME2` with a different index domain, `int(0..9)`:

`NAME2 : matrix indexed by [int(0..9)] of int(1..5)`

`NAME2` is also a 1-dimensional array with 10 Boolean elements, but is dereferenced differently: `NAME2[1]` dereferences the second element because its index domain starts with 0.

2.2 Basic Model Structure

An ESSENCE' model is structured in the following way:

1. Header with version number: `language Essence' 1.b.a`
2. Parameter declarations (optional)
3. Constant definitions (optional)
4. Variable declarations (optional)
5. Objective (optional)
6. Constraints (optional)

The version number needs a little explanation. Over time it is inevitable that the input language will change, and the basic reason for the version number is to ensure that the user is protected against changes to the format which might change the semantics. At all times Tailor should deal with a given version as intended, even if the latest version of the input format would be treated differently.¹ The version number is of the form `<number>.<letter1>.<letter2>`. The number corresponds to a major ESSENCE version number, and the first letter the major ESSENCE' number within that. Finally, the second letter indicates the minor ESSENCE' number.

Parameter declaration, Constant definitions and Variable declarations can be interleaved, but for readability we suggest to put them in the order given above. Comments are preceded by '\$'.

Parameter values are defined in a separate file, the *parameter file*. Parameter files have the same header as problem models and hold a list of parameter definitions. Table 1 gives an overview of the model structure of problem and parameter files. Each model part will be discussed in more detail in the following sections.

2.3 Constant Definitions

In most problem models there are re-occurring constant values and it can be useful to define them as constants. The `letting` statement allows to assign a name with a constant value. The statement

```
letting NAME be constant
```

introduces a new reserved name *NAME* that is associated with the constant value *constant*. Every subsequent occurrence of *NAME* in the model is replaced by *constant*. Please note that *NAME* cannot be used in the model *before* it has been defined. In the following subsections we discuss different kinds of constants.

¹In the worst case, a future version of Tailor may refuse to process an earlier language format if the changes are too drastic. We do not guarantee perfect emulation of earlier formats, for example if we implement bug fixes which affect how earlier versions are dealt with.

Problem Model Structure	Parameter File Structure
<pre>language ESSENCE' 1.b.a \$ parameter declaration given n : int \$ constant definition letting c be 5 \$ variable declaration find x,y : int(1..n) \$ constraints such that x + y >= c, x + c*y = 0</pre>	<pre>language ESSENCE' 1.b.a \$ parameter instantiation letting n be 7</pre>

Table 1: Model Structure of problem files and parameter files in ESSENCE'. '\$' denote comments.

2.3.1 Constant Expressions

The statement

```
letting  c be 10
```

introduces a new constant with name c that is assigned the value 10. Usually constants are written with lower-case letters. The constant expression may also contain other constants or parameter values, for instance with

```
letting  c be 10
letting  d be c*2
```

2.3.2 Constant Domains

Constant domains are defined in a similar way using the keywords `be domain`:

```
letting  INDEX be domain int(1..5)
```

defines a domain with name $INDEX$ that ranges from $int(1..5)$. For readability we suggest to use upper-case letters for domains. Constant domains may contain other constant/parameter values, such as in

```
letting  c          be          10
letting  INDEX     be domain  int(1..c)
```

2.3.3 Constant Arrays

Constant arrays are defined by stating a name, followed by the corresponding matrix type and the constant values in brackets. For instance,

```
letting name1 : matrix indexed by [ int(1..4) ] of int(1..10) be [2,8,5,9]
```

defines a 1-dimensional constant array $name_1$ with 4 Elements ranging from $int(1..10)$. Hence $name_1[3]$ is replaced by 5. 2-dimensional constant arrays are defined in the same way:

```
letting name2 :
    matrix indexed by [ int(1..2), int(1..4) ] of int(1..10)
    be [ [2,8,5,1],
        [3,7,9,4] ]
```

states that $name_2$ is a $(1..2) \times (1..4)$ array of integers ranging from $int(1..10)$ consisting of the values $[[2,8,5], [3,7,9]]$. Hence $name_2[1,2]$ is replaced by 8.

2.4 Parameter Declarations

Parameters are declared with the **given** statement followed by a domain the parameter ranges over. Parameters are allowed to range over the infinite domain int . As example, consider

```
given n : int
```

2.5 Variable Declaration

Variables are declared using **find** followed by a name and their corresponding domain. The example below

```
find x : int(1..10)
```

defines a variable x on the domain $int(1..10)$. It is possible to define several variables on the same domain as

```
find x, y, z : int(1..10)
```

that introduces 3 variables x, y, z ranging over $int(1..10)$. Arrays of variables are declared by defining the index domain and basedomain. Consider the example

```
find m : matrix indexed by [ int(1..10) ] of bool
```

that declares m as a 1-dimensional matrix of 10 Booleans.

2.6 Objective

The objective of a problem is either to **maximise** or **minimise** a variable or expression. For instance,

```
minimising x
```

states that the value assigned to variable x will be minimised. Objectives are written before the constraint definitions.

2.7 Constraints

After defining constants and declaring variables and parameters, constraints are specified with the keyword **such that**. ESSENCE' supports a wide range of operators (for more details on operators see Section 2.8):

- Basic Arithmetic Operators: `+` `-` `*` `/` `%` `|` `min` `max`
- Basic Boolean Operators: `\|` `/\` `=>` `<=>`
- Relational Operators: `=` `!=` `>` `<` `>=` `<=`
- Sum Operator: `sum`
- Quantification operators: `forall` `exists`
- Global Constraints: `alldifferent` `element`
- Table Constraint: `table`

We define two kinds of expressions: arithmetic and relational expressions. Arithmetic relations range over an integer domain, for instance $x + 3$ is an arithmetic expression ranging from $(lb(x) + 3..ub(x) + 3)$. Relational expressions range over the Boolean domain, for instance the relational expression $x = 3$ can either be *true* or *false*. Basic arithmetic operators and the `sum` operator produce arithmetic expressions and all other operators produce relational expressions.

Please note that each operator has a certain precedence and you will sometimes need to use parenthesis to express certain constraints. As an example, consider the expression

$$x = y \wedge y \leq z$$

According to the operator precedence in ESSENCE', the expression would be parsed as

$$x = ((y \wedge y) \leq z)$$

which might not be intended. Setting parenthesis is helpful to ensure a certain meaning, as illustrated below:

$$(x = y) \wedge (y \leq z)$$

More details about operator precedence is given in Section 2.8.

2.7.1 The sum Operator

The `sum` operator corresponds to the mathematical \sum and has the following syntax:

$$\text{sum } \textit{quantified-variable}(s) : \textit{domain} . \textit{expression}$$

For example, if we want to take the sum from 1 to 10 we write

```
sum i : int(1..10) . i
```

which corresponds to

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

Several quantified variables may be defined for a particular quantification domain and `sums` can also be nested:

```
sum i, j : int(1..10) .
  sum k : int(i..10) .
    x[i,j] * k
```

2.7.2 Universal and Existential Quantification

Universal and existential quantification are powerful means to write down a series of constraints in a compact way. Quantifications have the same syntax as `sum`, but with `forall` and `exists` as keywords:

```
forall quantified-variable(s) : domain . expression
```

For instance, the universal quantification

```
forall i : int(1..5) .
  x[i] = i
```

corresponds to the conjunction

$$(x[1] = 1) \wedge (x[2] = 2) \wedge \dots \wedge (x[5] = 5)$$

which could also be written as

$$\begin{aligned} x[1] &= 1, \\ x[2] &= 2, \\ &\dots \\ x[5] &= 5 \end{aligned}$$

An example for existential quantification is

```
exists i : int(1..5) .
  x[i] = i
```

that corresponds to the disjunction

$$(x[1] = 1) \vee (x[2] = 2) \vee \dots \vee (x[5] = 5)$$

Quantifications can range over several quantified variables and can be arbitrarily nested, as demonstrated with the `sum` operator.

Operator(s)	Functionality	Associativity
,	comma	Left
:	colon	Left
()	left and right parenthesis	Left
[]	left and right brackets	Left
!	not	Right
\&	and	Left
\	or	Left
=>	if (implication)	Left
<=>	iff (logical equality)	Left
-	unary minus	Right
^	power	Left
* /	multiplication, integer division	Left
+ -	addition, subtraction	Left
< <= > >=	(lex)less, (lex)less or equal,	none
<lex <=lex >lex >=lex	(lex)greater, (lex)greater or equal	
= !=	equality, disequality	none
.	dot	Right

Table 2: Operator precedence in ESSENCE'

2.8 Operators in ESSENCE'

Arithmetic operators and the sum operator return arithmetic expressions while all other expressions return relational expressions. Arithmetic operators may only be used on arithmetic expressions with the exception of addition and subtraction (i.e. Booleans may be added/subtracted).

To use operators correctly, you have to understand the precedence and associativity of operators. Table 2 describes the precedence and associativity of the operators, arranged by decreasing order of precedence (the operators on top have highest precedence).

As you would expect, operators with higher precedence take priority, so are applied first. We have, for example

$$a => b^c * d / (e + f) \equiv a => ((b^c) * d) / (e + f)$$

For associativity, an operator \cdot with left associativity has $a \cdot b \cdot c \equiv (a \cdot b) \cdot c$ while with right associativity we have $a \cdot b \cdot c \equiv a \cdot (b \cdot c)$. The operators with no associativity defined are meaningless if nested, so that $a = b = c$ is incorrect.

3 Examples

3.1 EightNumberPuzzle.eprime

```
language ESSENCE' 1.b.a
```

```
find circles: matrix indexed by [int(1..8)] of int(1..8)
```

```
such that
```

```
alldiff(circles),
| circles[1] - circles[2] | > 1,
| circles[1] - circles[3] | > 1,
| circles[1] - circles[4] | > 1,
| circles[2] - circles[3] | > 1,
| circles[3] - circles[4] | > 1,
| circles[2] - circles[5] | > 1,
| circles[2] - circles[6] | > 1,
| circles[3] - circles[5] | > 1,
| circles[3] - circles[6] | > 1,
| circles[3] - circles[7] | > 1,
| circles[4] - circles[6] | > 1,
| circles[4] - circles[7] | > 1,
| circles[5] - circles[6] | > 1,
| circles[6] - circles[7] | > 1,
| circles[5] - circles[8] | > 1,
| circles[6] - circles[8] | > 1,
| circles[7] - circles[8] | > 1
```

3.2 FarmersProblem.eprime

```
language ESSENCE' 1.b.a
```

```
find pigs, hens: int(0..7)
```

```
such that
```

```
pigs + hens = 7,
pigs * 4 + hens * 2 = 22
```

3.3 K4P2GracefulGraph.eprime

```
language ESSENCE' 1.b.a
```

```
find nodes : matrix indexed by [int(1..8)] of int(0..16),
      edges: matrix indexed by [int(1..16)] of int(1..16)
```

such that

```
|nodes[1] - nodes[2]| = edges[1],
|nodes[1] - nodes[3]| = edges[2],
|nodes[1] - nodes[4]| = edges[3],
|nodes[2] - nodes[3]| = edges[4],
|nodes[2] - nodes[4]| = edges[5],
|nodes[3] - nodes[4]| = edges[6],
```

```
|nodes[5] - nodes[6]| = edges[7],
|nodes[5] - nodes[7]| = edges[8],
|nodes[5] - nodes[8]| = edges[9],
|nodes[6] - nodes[7]| = edges[10],
|nodes[6] - nodes[8]| = edges[11],
|nodes[7] - nodes[8]| = edges[12],
```

```
|nodes[1] - nodes[5]| = edges[13],
|nodes[2] - nodes[6]| = edges[14],
|nodes[3] - nodes[7]| = edges[15],
|nodes[4] - nodes[8]| = edges[16],
```

```
alldiff(edges),
alldiff(nodes)
```

3.4 NQueensBool.eprime

```
language ESSENCE' 1.b.a
```

```
given n: int
```

```
find queens: matrix indexed by [int(1..n), int(1..n)] of bool
```

such that

```
forall i : int(1..n). ( sum j : int(1..n). queens[i,j] ) = 1,
forall i : int(1..n). ( sum j : int(1..n). queens[j,i] ) = 1 ,
```

```
forall i : int(1..n). forall j: int(1..n). forall k:int(1..n). forall l:int(1..n).
(
  (
    (|i - k| = |j - l| ) /\
    ((i != k) \/\ (j != l))
  )
)
```

```
=>
```

```

    (queens[i,j] + queens[k,1] <= 1)
)

```

3.5 NQueensColumn.eprime

```
language ESSENCE' 1.b.a
```

```
given n: int
```

```
find queens: matrix indexed by [int(1..n)] of int(1..n)
```

```
such that
```

```
alldiff(queens),
```

```
forall i : int(1..n). forall j : int(i+1..n). | queens[i] - queens[j] | != | i - j |
```

3.6 SENDMOREMONEY.eprime

```
language ESSENCE' 1.b.a
```

```
find S,E,N,D,M,O,R,Y : int(0..9)
```

```
such that
```

```
1000*S + 100*E + 10*N + D +
1000*M + 100*O + 10*R + E =
10000*M + 1000*O + 100*N + 10*E + Y,
```

```
alldiff([S,E,N,D,M,O,R,Y])
```

3.7 zebra.eprime

```
language ESSENCE' 1.b.a
```

```
$red = colour[1]
```

```
$green = colour[2]
```

```
$ivory = colour[3]
```

```
$yellow = colour[4]
```

```
$blue = colour[5]
```

```
$Englishman = nationality[1]
```

```
$Spaniard = nationality[2]
```

```
$Ukranian = nationality[3]
```

```
$Norwegian = nationality[4]
```

```
$Japanese = nationality[5]
```

```
$coffee = drink[1]
```

```
$tea = drink[2]
$milk = drink[3]
$orange juice = drink[4]
$Old Gold = smoke[1]
$Kools = smoke[2]
$Chesterfields = smoke[3]
$Lucky Strike = smoke[4]
$Parliaments = smoke[5]
$dog = pets[1]
$snails = pets[2]
$fox = pets[3]
$horse = pets[4]
```

```
find colour: matrix indexed by [int(1..5)] of int(1..5),
    nationality: matrix indexed by [int(1..5)] of int(1..5),
    drink: matrix indexed by [int(1..5)] of int(1..5),
    smoke: matrix indexed by [int(1..5)] of int(1..5),
    pets: matrix indexed by [int(1..5)] of int(1..5)
```

such that

```
$constraints needed as this is a logical problem where
$the value allocated to each position of the matrix represents position of house
alldiff(colour),
alldiff(nationality),
alldiff(drink),
alldiff(smoke),
alldiff(pets),
```

```
$There are five houses.
```

```
$No constraint covered by domain specification
```

```
$The Englishman lives in the red house
nationality[1] = colour[1],
```

```
$The Spaniard owns the dog.
nationality[2] = pets[1],
```

```
$Coffee is drunk in the green house.
drink[1] = colour[2],
```

```
$The Ukranian drinks tea.
nationality[3] = drink[2],
```

```
$The green house is immediately to the right of the ivory house.
```

colour[2] + 1 = colour[3],

\$The Old Gold smoker owns snails.
smoke[1] = pets[2],

\$Kools are smoked in the yellow house.
smoke[2] = colour[4],

\$Milk is drunk in the middle house.
drink[3] = 3,

\$The Norwegian lives in the first house
nationality[4] = 1,

\$The man who smokes Chesterfields lives in the house next to the man with the fox.
|smoke[3] - pets[3]| = 1,

\$Kools are smoked in the house next to the house where the horse is kept.
|smoke[2] - pets[4]| = 1,

\$The Lucky Strike smoker drinks orange juice.
smoke[4] = drink[4],

\$The Japanese smokes Parliaments.
nationality[5] = smoke[5],

\$The Norwegian lives next to the blue house.
|nationality[4] - colour[5]| = 1

3.8 zebraAlt.eprime

language ESSENCE' 1.b.a

\$red = colour[1]
\$green = colour[2]
\$ivory = colour[3]
\$yellow = colour[4]
\$blue = colour[5]
\$Englishman = nationality[1]
\$Spaniard = nationality[2]
\$Ukranian = nationality[3]
\$Norwegian = nationality[4]
\$Japanese = nationality[5]
\$coffee = drink[1]
\$tea = drink[2]
\$milk = drink[3]

```
$orange juice = drink[4]
$Old Gold = smoke[1]
$Kools = smoke[2]
$Chesterfields = smoke[3]
$Lucky Strike = smoke[4]
$Parliaments = smoke[5]
$dog = pets[1]
$snails = pets[2]
$fox = pets[3]
$horse = pets[4]
```

```
find colour: matrix indexed by [int(1..5)] of int(1..5),
    nationality: matrix indexed by [int(1..5)] of int(1..5),
    drink: matrix indexed by [int(1..5)] of int(1..5),
    smoke: matrix indexed by [int(1..5)] of int(1..5),
    pets: matrix indexed by [int(1..5)] of int(1..5)
```

such that

```
$constraints needed as this is a logical problem where
$the value allocated to each position of the matrix represents position of house
alldiff(colour),
alldiff(nationality),
alldiff(drink),
alldiff(smoke),
alldiff(pets),
```

```
$There are five houses.
```

```
$No constraint covered by domain specification
```

```
$The Englishman lives in the red house
nationality[1] = colour[1],
```

```
$The Spaniard owns the dog.
nationality[2] = pets[1],
```

```
$Coffee is drunk in the green house.
drink[1] = colour[2],
```

```
$The Ukrainian drinks tea.
nationality[3] = drink[2],
```

```
$The green house is immediately to the right of the ivory house.
colour[2] + 1 = colour[3],
```

\$The Old Gold smoker owns snails.
smoke[1] = pets[2],

\$Kools are smoked in the yellow house.
smoke[2] = colour[4],

\$Milk is drunk in the middle house.
drink[3] = 3,

\$The Norwegian lives in the first house
nationality[4] = 1,

\$The man who smokes Chesterfields lives in the house next to the man with the fox.
(smoke[3] = pets[3]+1) \ / (smoke[3] = pets[3]-1),

\$Kools are smoked in the house next to the house where the horse is kept.
(smoke[2] = pets[4]+1) \ / (smoke[2] = pets[4]-1),

\$The Lucky Strike smoker drinks orange juice.
smoke[4] = drink[4],

\$The Japanese smokes Parliaments.
nationality[5] = smoke[5],

\$The Norwegian lives next to the blue house.
(nationality[4] = colour[5]+1) \ / (nationality[4] = colour[5]-1)

References

[1] Java <http://java.sun.com>