ESSENCE PRIME Description

Peter Nightingale and Andrea Rendl

1 Introduction

The purpose of this document is to describe the ESSENCE PRIME language and to be a reference for users of ESSENCE PRIME. ESSENCE PRIME is a constraint modelling language, therefore it is mainly designed for describing $\mathcal{NP}$-hard decision problems. It is not the only (or the first) constraint modelling language. ESSENCE PRIME began as a subset of ESSENCE [1] and has been extended from there. It is similar to the earlier Optimization Programming Language (OPL) [4]. ESSENCE PRIME is implemented by the tool SAVILE ROW [2, 3].

ESSENCE PRIME is considerably different to procedural programming languages, it does not specify a procedure to solve the problem. The user specifies the problem in terms of decision variables and constraints, and the solver automatically finds a value for each variable such that all constraints are satisfied. This means, for example, that the order the constraints are presented in ESSENCE PRIME is irrelevant.

ESSENCE PRIME allows the user to solve *constraint satisfaction problems* (CSPs). A simple example of a CSP is a Sudoku puzzle (Figure 1). To convert a single Sudoku puzzle to CSP, each square can be represented as a decision variable with domain $\{1 \ldots 9\}$. The clues (filled in squares) and the rules of the puzzle are easily expressed as constraints.

We will use Sudoku as a running example. A simple first attempt of modelling Sudoku in ESSENCE PRIME is shown below. In this case the clues (for example `M[1,1]=5`) are included in the model. We have used not-equal constraints to state that all digits must be used in each row and column. We have also omitted the sub-square constraints for now.

```
language ESSENCE' 1.0

find M : matrix indexed by [int(1..9), int(1..9)] of int(1..9)

such that

M[1,1]=5,
M[1,2]=3,
M[1,5]=7,
....
M[9,9]=9,

forAll row : int(1..9) .
    forAll col1 : int(1..9) .
        forAll col2: int(col1+1..9) . M[row, col1]!=M[row, col2],

forAll col : int(1..9) .
    forAll row1 : int(1..9) .
        forAll row2: int(row1+1..9) . M[row1, col]!=M[row2, col]
```

In this example, some CSP decision variables are declared using a `find` statement. It is also worth noting that other variables exist that are not decision variables, for example, `row` is a *quantifier* variable that exists only to apply some constraints to every row.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 1: On the left is a Sudoku puzzle. The objective is to fill in all blank squares using only the digits 1-9 such that each row contains all the digits 1-9, each column also contains all the digits 1-9, and each of the 3 by 3 subsquares outlined in thicker lines also contains all digits 1-9. On the right is the solution. (Images are public domain from Wikipedia.)

An ESSENCE PRIME model usually describes a *class* of CSPs. For example, it might describe the class of all Sudoku puzzles. In order to solve a particular instance of Sudoku, the instance would be specified in a separate *parameter* file (also written in ESSENCE PRIME). The model would have parameter variables (of type integer, boolean or matrix), and the parameter file would specify a value for each of these variables.

Since ESSENCE PRIME is a constraint modelling language, we will define the constraint satisfaction problem (CSP). A CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is defined as a set of n *decision variables* $\mathcal{X} = \langle x_1, \ldots, x_n \rangle$, a set of domains $\mathcal{D} = \langle D(x_1), \ldots, D(x_n) \rangle$ where $D(x_i) \subsetneq \mathbb{Z}, |D(x_i)| < \infty$ is the finite set of all potential values of x_i, and a conjunction $\mathcal{C} = C_1 \wedge C_2 \wedge \cdots \wedge C_e$ of constraints.

For CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, a constraint $C_k \in \mathcal{C}$ consists of a sequence of $m > 0$ variables $\mathcal{X}_k = \langle x_{k_1}, \ldots, x_{k_m} \rangle$ with domains $\mathcal{D}_k = \langle D(x_{k_1}), \ldots, D(x_{k_m}) \rangle$ s.t. $\mathcal{X}_k$ is a subsequence[1] of $\mathcal{X}$, $\mathcal{D}_k$ is a subsequence of $\mathcal{D}$, and each variable x_{k_i} and domain $D(x_{k_i})$ matches a variable x_j and domain $D(x_j)$ in $\mathcal{P}$. C_k has an associated set $C_k^S \subseteq D(x_{k_1}) \times \cdots \times D(x_{k_m})$ of tuples which specify allowed combinations of values for the variables in $\mathcal{X}_k$.

2 The ESSENCE PRIME Expression Language

In ESSENCE PRIME *expressions* are built up from variables and literals using operators (such as +). We will start by describing the types that expressions may take, then describe the simplest expressions and build up from there.

2.1 Types and Domains

Types and domains play a similar role; they prescribe a set of values that an expression or 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. The type contains all integers, and the domain is a finite subset.

ESSENCE PRIME 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 PRIME is a finite-domain language; every decision variable is associated with a finite domain of values.

The atomic types of ESSENCE PRIME are int (integer) and bool (boolean). There is also a compound type, matrix, that is constructed of atomic types.

---

[1] I use subsequence in the sense that $\langle 1, 3 \rangle$ is a subsequence of $\langle 1, 2, 3, 4 \rangle$.

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

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

Integer Domains An integer domain represents a finite subset of the integers, and is specified as a sequence of ranges or individual elements, for example `int(1..3,5,7,9..20)`. Each range includes the endpoints, so the meaning of a range `a..b` is the set $\{i \in \mathbb{Z} | a \leq i \leq b\}$. A range `a..b` would normally be in order, i.e. `a<=b` but this is not strictly required. Out-of-order ranges correspond to the empty set of integers.

The meaning of an integer domain is the union of the ranges and individual elements in the domain. For example, `int(10, 1..5, 4..9)` is equivalent to `int(1..10)`.

Finally, the elements and endpoints of ranges may be expressions of type `int`. The only restriction is that they may not contain any CSP decision variables. The integer expression language is described in the following sections.

Matrix Domains A matrix is defined by the keyword `matrix`, followed by its dimensions and the base domain. The dimensions are a list, in square brackets, of domains. For instance,

```
Matrix1 : matrix indexed by [int(1..10),int(1..10)] of int(1..5)
```

is the domain of a two-dimensional square matrix, indexed by 1..10 in both dimensions, where each element of the matrix has the domain `int(1..5)`. Elements of this matrix would be accessed by `Matrix1[A,B]` where A and B are integer expressions.

Matrices may be indexed by any integer domain or the boolean domain. For example,

```
Matrix2 : matrix indexed by [int(-2..5),int(-10..10,15,17)] of int(1..5)
```

is a valid matrix domain.

2.2 Domain Expressions

ESSENCE PRIME contains a small expression language for boolean and integer domains. This language consists of three binary infix operators `intersect`, `union` and `-`. All three are left-associative and the precedences are given in Appendix A. The language also allows bracketed subexpressions with `(` and `)`.

For example, the first and second lines below are exactly equivalent.

```
letting dom be domain int(1..5, 3..8)
letting dom be domain int(1..5) union int(3..8)
```

2.3 Literals

Each of the three types (integer, boolean and matrix) has a corresponding literal syntax in ESSENCE PRIME. Any value of any type may be written as a literal. Sets and real numbers are not (as yet) part of the language. Integer and boolean literals are straightforward:

```
1 2 3 -5
true false
```

There are two forms of matrix literals. The simpler form is a comma-separated list of expressions surrounded by square brackets. For example, the following is a matrix literal containing four integer literals.

$$[1, 3, 2, 4]$$

Matrix literals may contain any valid expression in ESSENCE PRIME. For example a matrix literal is allowed to contain other matrix literals to build up a matrix with two or more dimensions. The types of the expressions contained in the matrix literal must all be the same.

The second form of matrix literal has an explicit index domain that specifies how the literal is indexed. This is specified after the comma-separated list of contents using a ; as follows.

$$[1, 3, 2, 4 ; \text{int}(4..6,8)]$$

The index domain must be a domain of type `bool` or `int`, and must contain the same number of values as the number of items in the matrix literal. When no index domain is specified, a matrix of size `n` is indexed by `int(1..n)`.

Finally a multi-dimensional matrix value can be expressed by nesting matrix literals. Suppose we have the following domain:

$$\text{matrix indexed by } [\text{int}(-2..0),\text{int}(1,2,4)] \text{ of int}(1..5)$$

One value contained in this domain is the following:

```
[ [ 1,2,3 ; int(1,2,4) ],
  [ 1,3,2 ; int(1,2,4) ],
  [ 3,2,1 ; int(1,2,4) ]
  ; int(-2..0) ]
```

2.4 Variables

Variables are identified by a string. Variable names must start with a letter `a-z` or `A-Z`, and after the first letter may contain any of the following characters: `a-z A-Z 0-9 _`. A variable may be of type integer, boolean or matrix.

Scoping of variables depends on how they are declared and is dealt with in the relevant sections below. As well as a type, variables have a category. The category is *decision*, *quantifier* or *parameter*. Decision variables are CSP variables, and the other categories are described below.

Expressions containing decision variables are referred to as *decision expressions*, and expressions containing no decision variables as *non-decision expressions*. This distinction is important because expressions in certain contexts are not allowed to contain decision variables.

2.5 Expression Types

Expressions can be of any of the three basic types (integer, boolean or matrix). Integer expressions range over an integer domain, for instance `x + 3` (where `x` is an integer variable) is an integer expression ranging from $lb(x) + 3$ to $ub(x) + 3$. Boolean expressions range over the Boolean domain, for instance the integer comparison `x = 3` can either be `true` or `false`.

2.6 Type Conversion

Boolean expressions or literals are automatically converted to integers when used in a context that expects an integer. As is conventional `false` is converted to `0` and `true` is converted to `1`. For example, the following are valid ESSENCE PRIME boolean expressions.

```
1+2-3+true-(x<y)=5
false < true
```

Integer expressions are not automatically converted to boolean. Matrix expressions cannot be converted to any other type (or vice versa).

2.7 Matrix Indexing and Slicing

Suppose we have a three-dimensional matrix `M` with the following domain:

```
matrix indexed by [int(1..3), int(1..3), bool] of int(1..5)
```

   `M` would be indexed as follows: `M[x,y,z]`, where `x` and `y` may be integer or boolean expressions and `z` must be boolean. Because the matrix has the base domain `int(1..5)`, `M[x,y,z]` will be an integer expression. Matrix indexing is a partial function: when one of the indices is out of bounds then the expression is undefined. ESSENCE PRIME has the relational semantics, in brief this means that the boolean expression containing the undefined expression is `false`. So for example, `M[1,1,false]=M[2,4,true]` is always `false` because the 4 is out of bounds. The relational semantics are more fully described in Section 4 below.

   Parts of matrices can be extracted by *slicing*. Suppose we have the following two-dimensional matrix named `N`:

```
[ [ 1,2,3 ; int(1,2,4) ],
  [ 1,3,2 ; int(1,2,4) ],
  [ 3,2,1 ; int(1,2,4) ]
  ; int(-2..0) ]
```

   We could obtain the first row by writing `N[-2,..]`, which is equal to `[ 1,2,3 ; int(1..3)]`. Similarly the first column can be obtained by `N[..,1]` which is `[ 1,1,3 ; int(1..3)]`. In general, the indices in a matrix slice may be `..` or an integer or boolean expression that does not contain any decision variables. Matrix slices are always indexed contiguously from 1 regardless of the original matrix domain.

   When a matrix slice has an integer or boolean index that is out of bounds, then the expression is undefined and this is handled as described in Section 4.

2.8 Integer and Boolean Expressions

ESSENCE PRIME has a range of binary infix and unary operators and functions for building up integer and boolean expressions, for example:

- Integer operators: `+ - * ** / % | min max`

- Boolean operators: `\/ /\ -> <-> !`

- Quantified sum: `sum`

- Logical quantifiers: `forAll exists`

- Numerical comparison operators: `= != > < >= <=`

- Matrix comparison operators: `<=lex <lex >=lex >lex`

- Set operator: `in`

- Global constraints: `allDiff gcc`

- Table constraint: `table`

These are described in the following subsections.

2.8.1 Integer Operators

ESSENCE PRIME has the following binary integer operators: `+ - * / % **`. `+`, `-` and `*` are the standard integer operators.

The operators `/` and `%` are integer division and modulo functions. `a/b` is defined as $\lfloor a/b \rfloor$ (i.e. it always rounds down). This does not match some programming languages, for example C99 which rounds towards 0.

The modulo operator `a%b` is defined as $a - b\lfloor a/b \rfloor$ to be complementary to `/`.

```
3/2 = 1
(-3)/2 = -2
3/(-2) = -2
(-3)/(-2) = 1

3 % 2 = 1
(-3) % 2 = 1
3 % (-2) = -1
(-3) % (-2) = -1
```

`**` is the power function: `x**y` is defined as x^y. There are two unary functions: absolute value (where `|x|` is the absolute value of `x`), and negation (unary `-`).

2.8.2 Boolean Operators

ESSENCE PRIME has the `/\` (and) and `\/` (or) operators defined on boolean expressions. There are also `->` (implies), `<->` (if and only if), and `!` (negation). These operators all take boolean expressions and produce a new boolean expression. They can be nested arbitrarily.

The comma `,` in ESSENCE PRIME is also a binary boolean operator, with the same meaning as `/\`. However it has a different precedence, and is used quite differently. `/\` is normally used within a constraint, and `,` is used to separate constraints (or separate groups of constraints constructed using a `forAll`). Consider the following example.

```
forAll i : int(1..n) . x[i]=y[i] /\ x[i]!=y[i+1],
exists i : int(1..n) . x[i]=1 /\ y[i]!=y[i+1]
```

Here we have two quantifiers, both with an `/\` inside. The comma is used to separate the `forAll` and the `exists`. The comma has the lowest precedence of all binary operators.

2.8.3 Integer and Boolean Functions

ESSENCE PRIME has named functions `min(X,Y)` and `max(X,Y)` that both take two integer expressions X and Y. `min` and `max` can also be applied to one-dimensional matrices to obtain the minimum or maximum of all elements in the matrix (see Section 2.10). `factorial(x)` returns the factorial of values from 0 to 20 where the result fits in a 64-bit signed integer. It is undefined for other values of x and the expression x is not allowed to contain decision variables. `popcount(x)` returns the bit count of the 64-bit two's complement representation of x, and x may not contain decision variables.

The function `toInt(x)` takes a boolean expression x and converts to an integer 0 or 1. This function is included only for compatibility with ESSENCE: it is not needed in ESSENCE PRIME because booleans are automatically cast to integers.

2.8.4 Numerical Comparison Operators

ESSENCE PRIME provides the following integer comparisons with their obvious meanings: `= != > < >= <=`. These operators each take two integer expressions and produce a boolean expression.

2.8.5   Matrix Comparison Operators

ESSENCE PRIME provides a way of comparing one-dimensional matrices. These operators compare two matrices using the dictionary order (*lexicographical* order, or lex for short).

There are four operators. A `<lex` B ensures that A comes before B in lex order, and A `<=lex` B which ensures that A `<lex` B or A=B. `>=lex` and `>lex` are identical but with the arguments reversed. For all four operators, A and B may have different lengths and may be indexed differently, but they must be one-dimensional. Multi-dimensional matrices may be flattened to one dimension using the `flatten` function described in Section 2.10 below.

2.8.6   Set Operator

The operator `in` states that an integer expression takes a value in a set expression. The set espression may not contain decision variables. The set may be a domain expression (Section 2.2) or the `toSet` function that converts a matrix to a set, as in the examples below.

```
x+y in (int(1,3,5) intersect int(3..10))
x+y in toSet([ i | i : int(1..n), i%2=0])
```

2.8.7   The Quantified Sum Operator

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

$$\text{sum i : D . E}$$

where `i` is a quantifier variable, `D` is a domain, and `E` is the expression contained within the `sum`. More than one quantifier variable may be created by writing a comma-separated list `i,j,k`.

For example, if we want to take the sum of all numbers in the range 1 to 10 we write

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

which corresponds to $\sum_{i=1}^{n} i$. n cannot be a decision variable.

Quantified sum has several similarities to the `forAll` and `exists` quantifiers (described below in Section 2.8.8): it introduces new local variables (named quantifier variables) that can be used within `E`, and the quantifier variables all have the same domain `D`. However `sum` has one important difference: a `sum` is an integer expression.

A quantified sum may be nested inside any other integer operator, including another quantified sum:

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

2.8.8   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  i : D . E
exists  i : D . E
```

For instance, the universal quantification

```
forAll i : int(1..3) . x[i] = i
```

corresponds to the conjunction:

```
x[1] = 1 /\ x[2] = 2 /\ x[3] = 3
```

An example of existential quantification is

```
exists i : int(1..3) . x[i] = i
```

and it corresponds to the following disjunction:

```
x[1] = 1 \/ x[2] = 2 \/ x[3] = 3
```

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

In the running Sudoku example, `forAll` quantification is used to build the set of constraints. The expression:

```
forAll row : int(1..9) .
   forAll col1 : int(1..9) .
       forAll col2: int(col1+1..9) . M[row, col1]!=M[row, col2]
```

is a typical use of universal quantification.

2.8.9 Quantification over Matrix Domains

All three quantifiers are defined on matrix domains as well as integer and boolean domains. For example, to quantify over all permutations of size `n`:

```
forAll perm : matrix indexed by [int(1..n)] of int(1..n) .
   allDiff(perm) -> exp
```

The variable `perm` represents a matrix drawn from the matrix domain, and the `allDiff` constraint evaluates to `true` when `perm` is a permutation of $1 \ldots n$. Hence the expression `exp` is quantified for all permutations of $1 \ldots n$.

If n is a constant, the example above could be written as a set of n nested `forAll` quantifiers. However if n is a parameter of the problem class, it is very difficult to write the example above using other (non-matrix) quantifiers.

2.8.10 Global Constraints

ESSENCE PRIME provides a small set of global constraints such as `allDiff` (which is satisfied when a vector of variables each take different values). Global constraints are all boolean expressions in ESSENCE PRIME. Typically it is worth using these in models because the solver often performs better with a global constraint compared to a set of simpler constraints.

For example, the following two lines are semantically equivalent (assuming x is a matrix indexed by `1..n`).

```
forAll i,j : int(1..n) . i<j -> x[i]!=x[j]
allDiff(x)
```

Both lines will ensure that the variables in x take different values. However the `allDiff` will perform better in most situations.[2] Table 1 summarises the global constraints available in ESSENCE PRIME.

Now we have the `allDiff` global constraint, the Sudoku example can be improved and simplified as follows:

```
language ESSENCE' 1.0

find M : matrix indexed by [int(1..9), int(1..9)] of int(1..9)

such that
```

---

[2]In the current version of Savile Row, with default settings, the `x[i]!=x[j]` constraints would be aggregated to create `allDiff(x)` therefore there is no difference in performance between these two statements. There would be a difference in performance when aggregation is switched off (for example by using the `-O1` flag).

Global Constraint	Arguments	Description
`allDiff(X)`	`X` is a matrix	Ensures expressions in `X` take distinct values in any solution.
`atleast(X, C, Vals)`	`X` is a matrix `Vals` is a matrix of non-decision expressions `C` is a matrix of non-decision expressions	For each non-decision expression `Vals[i]`, the number of occurrences of `Vals[i]` in `X` is at least `C[i]`.
`atmost(X, C, Vals)`	`X` is a matrix `Vals` is a matrix of non-decision expressions `C` is a matrix of non-decision expressions	For each non-decision expression `Vals[i]`, the number of occurrences of `Vals[i]` in `X` is at most `C[i]`.
`gcc(X, Vals, C)`	`X` is a matrix `Vals` is a matrix of non-decision expressions `C` is a matrix	For each non-decision expression `Vals[i]`, the number of occurrences of `Vals[i]` in `X` equals `C[i]`.
`alldifferent_except (X, Value)`	`X` is a matrix `Value` is a non-decision expression	Ensures variables in `X` take distinct values, except that `Value` may occur any number of times.

Table 1: Global constraints in ESSENCE PRIME. Each may be nested within expressions and have arbitrary expressions nested within them. Each matrix parameter of a global constraint must be a one-dimensional matrix. In some cases the parameter is a matrix of *non-decision expressions* – that is, integer or boolean expressions that contain no decision variables. Quantifier and parameter variables are allowed in non-decision expressions.

```
M[1,1]=5,
M[1,2]=3,
M[1,5]=7,
....
M[9,9]=9,

forAll row : int(1..9) .
    allDiff(M[row,..]),

forAll col : int(1..9) .
    allDiff(M[..,col])
```

Global constraints are boolean expressions like any other, and are allowed to be used in any context that accepts a boolean expression.

2.8.11 Table Constraints

In a table constraint the satisfying tuples of the constraint are specified using a matrix. This allows a table constraint to theoretically implement any relation, although practically it is limited to relations where the set of satisfying tuples is small enough to store in memory and efficiently search.

The first argument specifies the variables in the scope of the constraint, and the second argument is a two-dimensional matrix of satisfying tuples. For example, the constraint `a+b=c` on boolean variables could be written as a table as follows.

```
table( [a,b,c], [[0,0,0], [0,1,1], [1,0,1]] )
```

The first argument of `table` is a one-dimensional matrix expression. It may contain both decision variables and constants. The second argument is a two-dimensional matrix expression containing no decision variables. The second argument can be stated as a matrix literal, or an identifier, or by slicing a higher-dimension matrix, or by constructing a matrix using matrix comprehensions (see Section 2.9).

If the same matrix of tuples is used for many table constraints, a `letting` statement can be used to state the matrix once and use it many times. Lettings are described in Section 3.2 below.

2.9   Matrix Comprehensions

We have seen that matrices may be written explicitly as a matrix literal (Section 2.3), and that existing matrices can be indexed and sliced (Section 2.7). Matrices can also be constructed using *matrix comprehensions*. This provides a very flexible way to create matrices of variables or values. A single matrix comprehension creates a one-dimensional matrix, however they can be nested to create multi-dimensional matrices. There are two syntactic forms of matrix comprehension:

```
[ exp | i : domain1, j : domain2, cond1, cond2 ]
[ exp | i : domain1, j : domain2, cond1, cond2 ; indexdomain ]
```

where `exp` is any integer, boolean or matrix expression. This is followed by any number of comprehension variables, each with a domain. After the comprehension variables we have an optional list of conditions: these are boolean expressions that constrain the values of the comprehension variables. Finally there is an optional index domain. This provides an index domain to the constructed matrix.

To expand the comprehension, each assignment to the comprehension variables that satisfies the conditions is enumerated in lexicographic order. For each such assignment, the values of the comprehension variables are substituted into `exp`. The resulting expression then becomes one element of the constructed matrix.

The simplest matrix comprehensions have only one comprehension variable, as in the example below.

```
[ num**2 | num : int(1..5) ] = [ 1,4,9,16,25 ; int(1..5) ]
```

The matrix constructed by a comprehension is one-dimensional and is either indexed from 1 contiguously, or has the given index domain. The given domain must have a lower bound but is allowed to have no upper bound. For example the first line below produces the matrix on the second line.

```
[ i+j | i: int(1..3), j : int(1..3), i<j ; int(7..) ]
[ 3, 4, 5 ; int(7..9) ]
```

Matrix comprehensions allow more advanced forms of slicing than the matrix slice syntax in Section 2.7. For example it is possible to slice an arbitrary subset of the rows or columns of a two-dimensional matrix. The following two nested comprehensions will slice out the entries of a matrix `M` where both rows and columns are odd-numbered, and build a new two-dimensional matrix.

```
[ [ M[i,j] | j : int(1..n), j%2=1 ] | i : int(1..n), i%2=1 ]
```

Now we have matrix comprehensions, the Sudoku example can be completed by adding the constraints on the 3×3 subsquares. A comprehension is used to create a matrix of variables and the matrix is used as the parameter of an `allDiff` constraint.

```
language ESSENCE' 1.0

find M : matrix indexed by [int(1..9), int(1..9)] of int(1..9)

such that

M[1,1]=5,
M[1,2]=3,
M[1,5]=7,
....
M[9,9]=9,
```

Function	Arguments	Description
sum(X)	X is a one-dimensional matrix	Constructs the sum of elements in X
product(X)	X is a one-dimensional matrix	Constructs the product of elements in X
and(X)	X is a one-dimensional matrix of booleans	Constructs the conjunction of X
or(X)	X is a one-dimensional matrix of booleans	Constructs the disjunction of X
min(X)	X is a one-dimensional matrix	The integer minimum of elements in X
max(X)	X is a one-dimensional matrix	The integer maximum of elements in X
flatten(X)	X is a matrix	Constructs a one-dimensional matrix (indexed contiguously from 1) with the same contents as X
flatten(n,X)	X is a matrix	The first n+1 dimensions of X are flattened into one dimension that is indexed contiguously from 1. Therefore flatten(n,X) produces a new matrix with n fewer dimensions than X. The first argument n must be positive or 0, and flatten(0,X) returns X unchanged.
toSet(X)	X is a one-dimensional matrix of non-decision expressions	The set of elements in X

Table 2: Matrix Functions

```
forAll row : int(1..9) .
    allDiff(M[row,..]),

forAll col : int(1..9) .
    allDiff(M[..,col]),

$ all 3x3 subsquare have to be all-different
$ i,j indicate the top-left corner of the subsquare.
forAll i,j : int(1,4,7) .
    allDiff([ M[k,l]  | k : int(i..i+2), l : int(j..j+2)])
```

In this example, the matrix constructed by the comprehension depends on the values of i and j from the forAll quantifier. The comprehension variables k and l each take one of three values, to cover the 9 entries M[k,l] in the subsquare.

2.9.1 Matrix Comprehensions over Matrix Domains

Similarly to quantifiers, matrix comprehension variables can have a matrix domain. For example, the following comprehension builds a two-dimensional matrix where the rows are all permutations of 1..n.

```
[ perm | perm : matrix indexed by [int(1..n)] of int(1..n), allDiff(perm) ]
```

2.10 Functions on Matrices

Table 2 lists the matrix functions available in ESSENCE PRIME.

The functions sum, product, and and or were originally intended to be used with matrix comprehensions, but can be used with any matrix. The quantifiers sum, forAll and exists can be replaced with sum, and and or containing matrix comprehensions. For example, consider the forAll expression below (from the Sudoku model). It can be replaced with the second line below.

```
forAll row : int(1..9) . allDiff(M[row,..])
```

```
and([ allDiff(M[row,..]) | row : int(1..9) ])
```

In fact, matrix functions combined with matrix comprehensions are strictly more powerful than quantifiers. Also, the function `product` has no corresponding quantifier. Quantifiers are retained in the language because they can be easier to read.

As a more advanced example, given an $n \times n$ matrix M of decision variables, the sum below is the determinant of M using the Leibniz formula. The outermost comprehension constructs all permutations of $1 \ldots n$ using a matrix domain and an `allDiff`. Lines 3 and 4 contain a comprehension that is used to obtain the number of *inversions* of `perm` (the number of pairs of values that are not in ascending order). Finally line 5 builds a product of some of the entries of the matrix. Without the `product` function, it is difficult (perhaps impossible) to write the Leibniz formula in ESSENCE PRIME for a matrix of unknown size n.

```
sum([
    $  calculate the sign of perm from the number of inversions.
    ( (-1) ** sum([ perm[idx1]>perm[idx2]
                  | idx1 : int(1..n), idx2 : int(1..n), idx1<idx2 ]) )*
    product([ M[j, perm[j]] | j : int(1..n) ])
| perm : matrix indexed by [int(1..n)] of int(1..n), allDiff(perm)])
```

The `flatten` function is typically used to feed the contents of a multi-dimensional matrix expression into a constraint that accepts only one-dimensional matrices. For example, given a three-dimensional matrix M, the following example is a typical use of `flatten`.

```
allDiff( flatten(M[1,..,..]) )
```

When flattening a matrix M to create a new matrix F, the order of elements in F is defined as follows. Suppose M were written as a matrix literal (as in Section 2.3) the order elements are written in the matrix literal is the order the elements appear in F. The following example illustrates this for a three-dimensional matrix.

```
flatten([ [ [1,2], [3,4] ], [ [5,6], [7,8] ] ]) = [1,2,3,4,5,6,7,8]
```

3   Model Structure

An ESSENCE PRIME model is structured in the following way:

1. Header with version number: `language ESSENCE' 1.0`

2. Parameter declarations (optional)

3. Constant definitions (optional)

4. Decision variable declarations (optional)

5. Constraints on parameter values (optional)

6. Objective (optional)

7. Solver Control (optional)

8. Constraints

Parameter declaration, constant definitions and decision variable declarations can be interleaved, but for readability we suggest to put them in the order given above. Comments are preceded by '$' and run to the end of the line.

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 3 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.

Problem Model Structure	Parameter File Structure

`language ESSENCE' 1.0`
 	
$ *parameter declaration*
`given n :  int`
$ *constant definition*
`letting c=5`
 	
$ *variable declaration*
`find x,y :  int(1..n)`
 	
$ *constraints*
`such that`
` x + y >= c,`
` x + c*y = 0`
	`language ESSENCE' 1.0`
	
	$ *parameter instantiation*
	`letting n=7`

Table 3: Model Structure of problem files and parameter files in ESSENCE PRIME. '$' denotes comments.

3.1 Parameter Declarations with `given`

Parameters are declared with the `given` keyword followed by a domain the parameter ranges over. Parameters are allowed to range over the infinite domain `int`, or domains that contain an open range such as `int(1..)` and `int(..10)`. For example,

```
given n : int(0..)

given d : int(0..n)
```

declares two parameters, and the domain of the second depends on the value of the first. Parameters may have integer, boolean or matrix domains.

Now we have parameters we can generalise the Sudoku model to represent the problem class of all Sudoku puzzles. The parameter is the `clues` matrix, where blank spaces are represented as `0` and non-zero entries in `clues` are copied to `M`.

```
language ESSENCE' 1.0

given clues : matrix indexed by [int(1..9), int(1..9)] of int(0..9)

find M : matrix indexed by [int(1..9), int(1..9)] of int(1..9)

such that

forAll row : int(1..9) .
    forAll col : int(1..9) .
        (clues[row, col]!=0) -> (M[row, col]=clues[row, col]),

forAll row : int(1..9) .
    allDiff(M[row,..]),

forAll col : int(1..9) .
    allDiff(M[..,col]),
```

```
$ all 3x3 subsquare have to be all-different
$ i,j indicate the top-left corner of the subsquare.
forAll i,j : int(1,4,7) .
    allDiff([ M[k,l]  | k : int(i..i+2), l : int(j..j+2)])
```

3.2   Constant Definitions with `letting`

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 = A
```

introduces a new identifier `NAME` that is associated with the expression `A`. Every subsequent occurrence of `NAME` in the model is replaced by the value of `A`. Please note that `NAME` cannot be used in the model *before* it has been defined. `A` may be any integer, boolean or matrix expression that does not contain decision variables. Some integer examples are shown below.

```
given n : int(0..)
letting c = 10
letting d = c*n*2
```

Here the second integer constant depends on the first. As well as integer and boolean expressions, lettings may contain matrix expressions, as in the example below. When using a matrix literal the domain is optional – the two lettings below are equivalent. The version with the matrix domain may be useful when the matrix is not indexed from 1.

```
letting cmatrix = [ [2,8,5,1], [3,7,9,4] ]
letting cmatrix2 : matrix indexed by [ int(1..2), int(1..4) ] of int(1..10)
    = [ [2,8,5,1], [3,7,9,4] ]
```

Finally new matrices may be constructed using a slice or comprehension, as in the example below where the letting is used for the table of a table constraint.

```
letting xor_table = [ [a,b,c] | a : bool, b : bool, c : bool,
                                (a /\ b) \/ (!a /\ !b) <-> c ]
find x,y,z : bool
such that
table([x,y,z], xor_table)
```

3.2.1   Constant Domains

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

```
letting c = 10
given n : int(1..)
letting INDEX be domain int(1..c*n)
```

In this example `INDEX` is defined to be an integer domain, the upper bound of which depends on a parameter `n` and another constant `c`.

Constant domains are convenient when a domain is reused several times. In the Sudoku running example, we could use a letting for the domain `int(1..9)`:

```
language ESSENCE' 1.0
letting range be domain int(1..9)

given clues : matrix indexed by [range, range] of int(0..9)

find M : matrix indexed by [range, range] of range

such that

forAll row : range .
    forAll col : range .
        (clues[row, col]!=0) -> (M[row, col]=clues[row, col]),

forAll row : range .
    allDiff(M[row,..]),

forAll col : range .
    allDiff(M[..,col]),

$ all 3x3 subsquare have to be all-different
$ i,j indicate the top-left corner of the subsquare.
forAll i,j : int(1,4,7) .
    allDiff([ M[k,l]  | k : int(i..i+2), l : int(j..j+2)])
```

3.3 Decision Variable Declaration with `find`

Decision variables are declared using the `find` keyword followed by a name and their corresponding domain. The domain must be finite. The example below

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

defines a single decision variable with the given domain. It is possible to define several variables in one `find` by giving multiple names, as follows.

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

Matrices of decision variables are declared using a matrix domain, as in the following example.

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

This declares `m` as a 1-dimensional matrix of 10 boolean variables. Simple and matrix domains are described in Section 2.1.

In the Sudoku running example, we have been using the following two-dimensional matrix domain.

```
find M : matrix indexed by [int(1..9), int(1..9)] of int(1..9)
```

3.4 Constraints on Parameters with `where`

In some cases it is useful to restrict the values of the parameters. This is achieved with the `where` keyword, which is followed by a boolean expression containing no decision variables. In the following example, we require the first parameter to be less than the second.

```
given x : int(1..)
given y : int(1..)
where x<y
```

3.5 Objective

The objective of a problem is either `maximising` or `minimising` an integer or boolean expression. For instance,

```
minimising x
```

states that the value assigned to variable x will be minimised. Only one objective is allowed, and it is placed after all `given`, `find` and `letting` statements.

3.6 Solver Control

In addition to instructing the solver to minimise or maximise some expression, ESSENCE PRIME also supports some rudimentary options for controlling which variables the solver will branch on, and which variable ordering heuristic it will use. The information is passed on only when Minion is used as the solver. These statements are experimental and may be removed from the language in future versions.

The `branching on` statement specifies a sequence of variables for the solver to branch on. The `heuristic` statement specifies the heuristic used on only the variables in the `branching on` list. `heuristic` is followed by `static`, `sdf`, `conflict` or `srf` and these options are passed through to Minion.

The example below tells the solver to branch on `w` and `x` using the smallest domain first heuristic. It will subsequently branch on all the other decision variables using the default (static) ordering.

```
find w,x,y,z : int(1..10)
branching on [w,x]
heuristic sdf
```

The `branching on` statement is followed by a comma-separated list of individual decision variables or matrices. This list may contain matrices of different dimensions and sizes. Decision variables in matrices are enumerated in the order produced by the `flatten` function.

The `branching on` list may contain the same decision variable more than once. This can be useful to pick some variables from a matrix to branch on first, then include the rest of the matrix simply by including the entire matrix. In the following example the diagonal of `M` is branched first, then the rest of `M` is included.

```
find M : matrix indexed by [int(1..9), int(1..9)] of int(1..9)

branching on [ [ M[i,i] | i : int(1..9)], M ]
```

When a matrix is included in the `branching on` list, it is converted to a one-dimensional matrix using `flatten`.

Optimisation is only performed on the variables in the `branching on` list. Using `branching on` can cause `maximising` and `minimising` to function in an unusual way: they will only maximise or minimise on the variables in the `branching on` list, and therefore may not return the overall maximal/minimal solution.

3.7 Constraints

After defining constants and declaring decision variables and parameters, constraints are specified with the keywords `such that`. The constraints in ESSENCE PRIME are boolean expressions as described in Section 2.8.

Typically the constraints are written as a list of boolean expressions separated by the `,` operator.

4 Undefinedness in ESSENCE PRIME

Since the current version of ESSENCE PRIME is a closed language, there are a finite set of partial functions in the language. For example, `x/y` is a partial function because it is not defined when `y=0`. In its current version ESSENCE PRIME

implements the relational semantics as defined by Frisch and Stuckey (The Proper Treatment of Undefinedness in Constraint Languages, in Proc. Principles and Practice of Constraint Programming - CP 2009, pages 367-382). The relational semantics has the advantage that it can be implemented efficiently.

The relational semantics may be summarised as follows:

- Any integer or matrix expression directly containing an undefined expression is itself undefined.

- Any domain or domain expression directly containing an undefined expression is itself undefined.

- Any statement in the preamble (`find`, `letting` etc) directly containing an undefined expression is undefined.

- Any boolean expression that directly contains an undefined expression is `false`.

Informally, the relational semantics confines the effect of an undefined expression to a small part of the problem instance (which becomes `false`), in many cases avoiding making the entire problem instance `false`.

Consider the four examples below. Each contains a division by zero which is an undefined integer expression. In each case the division is contained in a comparison. Integer comparisons are boolean expressions.

```
(x/0 = y) = false
(x/0 != y) = false
! (x/0 = y) = true
! (x/0 != y) = true
```

Applying the rules of the relational semantics results in each of the comparisons inside the brackets becoming `false`:

```
(false) = false
(false) = false
! (false) = true
! (false) = true
```

In the relational semantics, `(x/0 != y)` is not semantically equivalent to `!(x/0 = y)`, which is somewhat counter-intuitive.

Another counter-intuitive case arises with matrix indexing. In the following example, the expression `M[0]` is undefined because 0 is not in the index domain. If the matrix is boolean (i.e. `DOM` is `bool`) then `M[0]` becomes `false`, and the model has a solution when `M[1]=false`. However, if the matrix contains integer variables (i.e. `DOM` is `int(0..1)`) then the constraint `M[0] = M[1]` becomes `false` and the model has no solutions.

```
find M : matrix indexed by [int(1)] of DOM
such that
M[0] = M[1]
```

In the SAVILE ROW implementation of ESSENCE PRIME, all partial functions are removed in a two-step process, before any other transformations are applied. The first step is as follows. For each partial function a boolean expression is created that is `true` when the partial function is defined and `false` when it is undefined. There are six operators that may be partial: division, modulo, power, factorial, matrix indexing and matrix slicing. Table 4 shows the generated boolean expression for each operator. The boolean expression is then added to the model by connecting it (with $/\backslash$) onto the closest boolean expression above the partial function in the abstract syntax tree.

The second step is to replace the partial function `P` with a total function `SP`. For each input where `P` is defined, `SP` is defined to the same value. For inputs where `P` is undefined, `SP` takes a default value (typically `0` for integer expressions).

Once both steps have been applied to each partial function, the model is well defined everywhere. This is done first, before any other model transformations, and thus allows all subsequent transformations to be simpler because there is no need to allow for undefinedness. For example, the expression `x/y=x/y` may not be simplified to `true`, because it is `false` when `y=0`. However, after replacing the partial division function, the resulting equality can be simplified to `true`. In general any equality between two syntactically identical expressions can be simplified to `true` once there are no partial functions.

Partial Function	Defined When
`X/Y`	`Y!=0`
`X%Y`	`Y!=0`
`X**Y`	`(X!=0 \/ Y!=0) /\ Y>=0`
`factorial(X)`	`X>=0`
Matrix indexing: `M[I,J,K]` where domain of `M` is `matrix indexed by [D1, D2, D3] of DBase`	`I ∈ D1 /\` `J ∈ D2 /\` `K ∈ D3`
Matrix slicing: `M[I,..,K]` where domain of `M` is `matrix indexed by [D1, D2, D3] of DBase`	`I ∈ D1 /\` `K ∈ D3`

Table 4: Partial functions in ESSENCE PRIME. `X`, `Y`, `I`, `J` and `K` are arbitrary expressions of the correct type.

A Operator Precedence in ESSENCE PRIME

Table 5 shows the precedence and associativity of operators in ESSENCE PRIME. As you would expect, operators with higher precedence are applied first.

Left-associative operators are evaluated left-first, for example `2/3/4 = (2/3)/4`. The only operator with right associativity is `**`. This allows double exponentiation to have its conventional meaning: `2**3**4 = 2**(3**4)`

Unary operators usually have a higher precedence than binary ones. There is one exception to this rule: that `**` has a higher precedence than unary minus. This allows `-2**2**3` to have its conventional meaning of `-(2**(2**3))=-256`, as opposed to `(-2)**(2**3)=256`.

B Reserved Words

The following words are keywords and therefore are not allowed to be used as identifiers.

```
forall, forAll, exists, sum,
such, that, letting, given, where, find, language,
int, bool, union, intersect, in, false, true
```

References

[1] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Mart'inez Hernández, and Ian Miguel. ESSENCE: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[2] Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, and Ian Miguel. Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In *20th International Conference on Principles and Practice of Constraint Programming (CP 2014)*, pages 590–605. Springer, 2014.

[3] Peter Nightingale, Patrick Spracklen, and Ian Miguel. Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP 2015)*, pages 330–340, 2015.

[4] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, Cambridge, MA, USA, 1999.

Operator	Functionality	Associativity	Precedence
!	Boolean negation		20
||	Absolute value		20
**	Power	Right	18
-	Unary negation		15
*	Multiplication	Left	10
/	Division	Left	10
%	Modulo	Left	10
intersect	Domain intersection	Left	2
union	Domain union	Left	1
+	Addition	Left	1
-	Subtraction & Domain Subtraction	Left	1
=	Equality	Left	0
!=	Disequality	Left	0
<=	Less-equal	Left	0
<	Less than	Left	0
>=	Greater-equal	Left	0
>	Greater than	Left	0
<=lex	Lex less-equal	Left	0
<lex	Lex less than	Left	0
>=lex	Lex greater-equal	Left	0
>lex	Lex greater than	Left	0
in	Value in a set	Left	0
/\	And	Left	-1
\/	Or	Left	-2
->	Implication	Left	-4
<->	If and only if	Left	-4
forAll, exists, sum	Quantifiers		-10
,	And	Left	-20

Table 5: Operator precedence in ESSENCE PRIME