Specification of the DOMINION Input Language Version 0.1

Ian P. Gent, Chris Jefferson, Lars Kotthoff, Ian Miguel, Peter Nightingale

November 9, 2009

1 Introduction

This document specifies the input language for the DOMINION constraint solver synthesiser. This language is relatively low-level since it is intended to be the input for model analysis. It does, however, support the specification of problem *classes*, as well as problem instances. Its syntax is substantially influenced by that of ESSENCE and ESSENCE′ [1, 2] but there are a number of differences, particularly in arrays and in DOMINION's support for set and list comprehensions.

2 Processing Stages and Instantiation Categories

We identify two important processing stages of a constraint model: 1. *Instantiation*, in which a value is given for every parameter in the model, defining an individual problem instance, and 2. *Solution*, a search for an assignment of values to each decision variable such that all constraints are satisfied.

The type system of the DOMINION input language distinguishes two *instantiation categories* related to the two processing stages: *parameter expressions* and *decision variable expressions*. *Parameter expressions* may contain constants, quantified variables and parameters, but no decision variables. Hence, the value of a parameter expression can be determined during instantiation. *Decision variable expressions* may contain parameter expressions and decision variables. Hence, the value of a variable expression is generally determined during the solution stage.

3 Types and Domains

In the DOMINION input language every expression has an immutable type, and the types of all expressions can be inferred and checked for correctness. It is also a finite-domain language: every decision variable is associated with a finite domain of values. We follow ESSENCE in regarding types and domains as distinct, though closely related, concepts. 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. Hence, each

domain is associated with an underlying type. At present, there are two atomic types, integers (defined as $\mathbb{Z}$) and sets of integers, and an array type constructor, which is used to build arrays of the atomic types of arbitrary dimension. Example domains include: the integers between 1 and 5, the sets of integers drawn from the interval 1..10.

3.1 Type Declaration

The type of an identifier is declared as it is introduced. For the atomic types the syntax is `int` or `set of int`. For arrays, the type is declared using a `dim` (for "dimension") statement as follows (n is a parameter):

```
dim firstArray[10,n]:  int
dim secondArray[2,2,n]:  set of int
```

Arrays are indexed from 0 in each dimension. A comma-separated list of parameter expressions within the square brackets indicates the number of dimensions of the array and the size of each. Following a `dim` declaration of an array, its elements are considered uninitialised and, therefore, empty. They are initialised individually, as shown below. In order to support the construction of triangular, or other non-rectangular arrays, it is **not** required that all elements of an array are initialised. It is, however, an error to initialise the same element of an array twice.

The alternative (and standard) to this approach to array declaration and initialisation is to declare rectangular arrays with uniform elements and then add constraints to approximate a non-rectangular array or to modify the elements so as to be non-uniform. This has the drawback of cluttering the model with these extra constraints, and can also lead to inefficiency. One example is if the constraint solver allocates backtrackable memory for decision variables most of whose domains are immediately pruned by unary constraints. Another is if the constraint solver has a special variable type that could have been used, had the 'true' domain been obvious (e.g. if some variables in the domain are, in fact, 0/1).

3.2 Domain Declaration

Domains are declared by suffixing the type declaration with a set (written using a 'set of ranges' style — see Section 7) denoting the subset of the given type from which the associated identifier takes its value. For example:

```
int {1,3..7}
set of int {1..49, 51..100}
```

The former indicates the domain of integers from 1 to 7, excluding 2. The latter indicates the domain of sets of integers drawn from 1 to 100, excluding 50. Domains are finite sets except in the case of parameters, in which case they may be unbounded, as explained in the following sub-section. As shown above, domain information is **not** given in the `dim` declaration of an array, but on an individual basis thereafter.

3.3 Parameters

Parameters are introduced with `given` statements. Their domains indicate the allowed parameter values in valid instances of the problem class. Examples include:

```
given a:  int
given b:  int {1..}
given c:  set of int {1..b}
given d[a, b]:  int
given e[f, g]:  set of int {1..50}
```

The parameter a's domain is all of the integers. This does not violate our finite-domain constraint solving assumption, since a has an individual finite value for any particular instance. Similarly, b is a parameter whose domain is the natural numbers. Hence, only instances in which a natural number is given for b are valid. The parameter c is a set whose domain depends on b, which is valid since b was declared previously.

The parameter d is a two-dimensional array whose size is determined by a and b (this implicitly constrains a to be positive). The parameter e is also a two-dimensional array, each element of which is a set drawn from 1..50. Since f and g were not declared previously, they are implicitly declared in this statement and their values are determined by the size of the given array for e in a particular instance.

3.4 Decision Variables

Decision variables are introduced with `find` statements. Examples include:

```
find var :  int {1..5, 7}
find var2 :  set of int {i+j | i in {1..4}, j in {1..4}}
```

Note that a finite domain is a requirement for decision variables. Set comprehension is covered in Section 8 below.

The elements of an array of decision variables can be initialised singly, or multiply as follows:

```
dim varArray[5,5]:  int
find varArray[4,0]:  int {1..4}
find varArray[2..,1]:  int {0..1}
[ find varArray[i,j]:  int {0..2} | i in {0..4}, j in {0..4}, i <= j]
```

The first statement following the `dim` statement above initialises `varArray`[4,0] to a decision variable whose domain is the integers between 1 and 4. The second initialises the elements `varArray`[2,1], `varArray`[3, 1], `varArray`[4,1] all to 0/1 decision variables. The last initialises a triangular array of decision variables, all of whose domains are the integers from 0 to 2. The format of list comprehensions is also covered in Section 8.

4 Constituents and File Structure

In order to support problem classes, the DOMINION input language takes the common step of separating model and instance data. DOMINION input is therefore split into two parts, a model file and zero or more associated parameter files.

A DOMINION model file has two parts:

1. The *preamble* in which parameters, constants, decision variables and arrays are defined.
2. The *constraints* over the decision variables.

A DOMINION parameter file gives values for each of the parameters in the associated model file.

5 The Preamble

The first line of the preamble, and hence a model file, specifies the version of the language, with a major and minor version number.

```
language Dominion X.Y
```

This is followed by interleaved `given`, `letting`, `find` and `dim` statements, which respectively introduce parameters, bind parameter expressions to identifiers, introduce decision variables and introduce and dimension arrays. The order of these statements is significant, and identifiers must be declared before use. All but `letting` statements were introduced in the previous section. These are explained below.

The final (and optional) part of the preamble deals with optimisation problems. The `minimize` (or `minimise`) and `maximize` (or `maximise`) statements allow a single decision variable to be minimized or maximized. Only one `minimize` or `maximize` statement is allowed, and it must be at the end of the preamble. The following statement minimizes a variable in an array.

```
minimize Arr[3,6,7]
```

5.1 Binding Parameter Expressions to Identifiers

The `letting` statement is used to declare identifiers for parameter expressions. Examples include:

```
letting a be 5
letting b be (15 * a)+1
letting c be {1,2,5..9}
letting d be {i*j | i in {1..5}, j in {1..5}, i<j}
```

Any valid arithmetic or set parameter expression (both detailed below) may be bound to an identifier in this way.

Arrays may also be declared to be composed of parameter expressions, rather than decision variables using the `const` keyword, as follows:

```
dim paramArray[5,5]:  const int
```

Note that an array can contain decision variables or parameter expressions, but **not** both. Similarly to arrays of decision vaiables, the elements of `paramArray` can be initialised singly, or multiply as follows:

```
 letting paramArray[0,0] be 4
letting paramArray[..,1] be 23
[ letting paramArray[i,j] be n | i in {0..4}, j in {0..4}, i == j]
```

The first statement above initialises `paramArray`[0,0] to the constant 4. The second initialises the elements `paramArray`[0,1], `paramArray`[1, 1], ..., `paramArray`[4,1] all to be the constant 23. The third initialises the elements of the leading diagonal of `paramArray` to the parameter n, which must have been declared previously.

6 The Constraints

The DOMINION input language specifies only the allowed syntax of constraints, not the set of supported constraints itself. This set is covered in a separate document (to be written!), which we expect to evolve quickly. Hence, this section contains predominantly BNF rather than concrete examples. The BNF for specifying constraints is as follows:

```
<Constraints> ::= "such that" (<Constraint>)*
<Constraint>  ::= "cons" <IdentifierString>
                      (<CtComprehension> | <ConstraintExpression>)
```

The `<IdentifierString>` binds a unique identifier to the constraint or constraint comprehension. A constraint comprehension is a comprehended constraint (cf. Section 8).

A `<ConstraintExpression>` is defined as follows:

```
<ConstraintExpression> ::= <IdentifierString>
                             "(" [<ConstraintArg>
                                 ("," <ConstraintArg>)* ] ")"
```

The `<IdentifierString>` specifies the name of the constraint type, e.g. `sum` or `product`. The constraint name is followed by a possibly empty list of `<ConstraintArg>`s in parentheses. A `<ConstraintArg>` can be a constant, a variable, or a constraint. It can also be a comma-separated list of each of these enclosed in square brackets or a variable comprehension. Finally, a `<ConstraintArg>` may also be another constraint to allow for things like reified constraints.

7 Expressions

We give a brief overview of the kinds of expressions that are allowed in the Dominion input language.

7.1 Arithmetic expressions

Valid arithmetic expressions are of type `int` and may contain symbols of type `int`, plain integers, and the usual arithmetic infix operators +, -, *, /. Minus is also allowed as a unary prefix operator. Any subexpression may be enclosed in parentheses ().

7.2 Unbounded set of ranges

An unbounded set of ranges represents a (possibly infinite) subset of `int` using ranges (e.g. 5..10) or individual numbers separated by commas, in ascending order. It is allowed to have .. at the beginning or end of the sequence for infinite sets. For example, the set of integers except 6 is expressed as $..5, 7...$

7.3 Bounded set of ranges

A bounded set of ranges represents a finite subset of the integers. It is the same as unbounded set of integers, but .. is not allowed at the beginning or end of the sequence.

7.4 Set Comprehensions

Set comprehensions are an alternative for specifying finite sets of integers. They are described in Section 8.3 below. The following is an example of a set comprehension representing {66}.

```
{ i+10*j | i in {1..6}, j in {6..10}, i==j }
```

8 Comprehensions

Comprehensions are used to write multiple similar `letting` and `find` statements, multiple similar constraints, to construct un-named arrays of variables and to construct sets of integers. They have the following basic form.

```
[ left-hand side | right-hand side ]
```

The right-hand side specifies a set of *comprehension parameters* and *comprehension conditions* (described in the next section). The left-hand side is an expression to be repeated with different values substituted for comprehension parameters.

8.1 Comprehension parameters and conditions

All three types of comprehension have a sequence of *comprehension parameters*. A comprehension parameter is a universally-quantified integer variable with a finite set of values.

For example, the following set comprehension constructs a set of the integers from 1 to n except 5 (where n must be a parameter expression). i is essentially a local universally-quantified variable which takes each value in its (finite) range.

```
{ i | i in {1..n}, i != 5 }
```

A comprehension may have any number of comprehension parameters, followed by any number of conditions. The only restriction is that it is not allowed to have no parameters *and* no conditions. The BNF for the right hand side of a comprehension is given below. Each parameter has a bounded constant set of values associated with it.

```
<CompParamsAndConditions> ::= <RangeList> ["," <ConditionList>] |
                              <ConditionList>

<RangeList> ::= <ParameterRange> ["," <RangeList>]
<ParameterRange> ::= <IdentifierString> "in" <BoundedConstantSet>
```

Comprehension conditions consist of two arithmetic expressions with a binary comparison operator between them. Both arithmetic expressions must be parameter expressions.

```
<ConditionList> ::= <ComprehensionCondition> ["," <ConditionList>]
<ComprehensionCondition> ::= <ArithExpr>
                             ("!=" | "<" | ">" | "<=" | ">=" | "==")
                             <ArithExpr>
```

Comprehensions are unrolled as follows. Assume the comprehension has the sequence of parameters $\langle x_1, \ldots, x_n \rangle$ (in the order they are specified), and each parameter x_i is assigned a value a_i (within its bounded set). An assignment of all parameters is contained in a sequence $\langle a_1, \ldots, a_n \rangle$. The space of these sequences is enumerated in lexicographic (dictionary) order. For each sequence of values which satisfies the comprehension conditions, an instance of the left-hand side of the comprehension is generated by substituting in the values.

8.2 Comprehensions of `letting` and `find`

In some situations it is desirable to generate lists of similar `letting` or `find` statements. This is done with comprehensions. For example, the following statements define an array `Pairs` which contains a decision variable for every unordered pair of integers in the range $\{0 \ldots n-1\}$.

```
given n : int {1..}
dim Pairs[n,n]: int
[ find Pairs[i,j] : int {1..5} | i in {0..n-1}, j in {0..n-1}, i<=j ]
```

All comprehension parameters (in this case i and j) must appear in the `letting` or `find` statement as indices in the left-hand side expression. In the example above, the left-hand side expression is `Pairs[i,j]`, and both comprehension parameters i and j appear as indices. Otherwise, the rules regarding `letting` and `find` apply unchanged.

The left-hand side of the `letting` or `find` statement must be an array symbol followed by indices. A statement like the following is not allowed:

```
[ find x : int{1..5} | a<b ]
```

In this case, the identifier x may be defined in some instantiations and not others. We forbid this to simplify analysis at the problem class level.

8.3 Set comprehensions

Set comprehensions allow a set of integers to be specified compactly. They are distinguished from other types of comprehension by using curly braces instead of square brackets. The following example is the set of even non-negative numbers up to n (where n is a parameter expression).

```
{ 2*i | i in {0..n/2} }
```

It would be impossible to specify the set of even numbers up to n in the set of ranges notation, therefore set comprehensions increase the expressiveness of the language.

It is permitted to have comprehension parameters that are not used in the left-hand side of the comprehension. Set comprehensions may only be used on the right-hand side of a `letting`, to initialize a `set of int`. Set comprehensions are always parameter expressions.

A set comprehension can be used to specify the domain of a decision variable. First, a `letting` is used to create an identifier for the domain, then `find` uses that identifier, as follows.

```
letting dom be { 2*i | i in {0..n/2} }
find x : int dom
```

8.4 Variable comprehensions

A variable comprehension creates an unnamed array of variables. For example, the following will construct an array containing elements on the diagonal of Arr.

```
[ Arr[i,i] | i in {0..n-1} ]
```

It is possible to nest variable comprehensions to create multi-dimensional arrays. For example, the following copies part of the two-dimensional array `Arr`. It is equivalent to `Arr[2..4,2..4]`.

```
[ [ Arr[i,j] | j in {2..4} ] | i in {2..4} ]
```

Variable comprehensions may only be used as parameters of constraints.

References

[1] A.M. Frisch, W. Harvey, C. Jefferson, B. Martinez-Hernandez, I. Miguel. Essence: A Constraint Language for Specifying Combinatorial Problems. *Constraints* 13(3), 268-306, 2008.

[2] A. Rendl. *Effective Compilation of Constraint Models*, PhD Thesis, University of St Andrews (to appear), 2010.

A BNF Grammar

```
<DominionModel> ::= <Preamble> <Constraints>

<Preamble> ::= "language" "Dominion" <Version>
               (<Given> | <Letting> | <Find> | <LetFindComprehension> | <Dim>)*
               [<Minimizing> | <Maximizing>]

<Version> ::= <Integer> "." <Integer>

<Dim> ::= "dim" <IdentifierString> <SizeMatrixExpr> "of" ["set" "of"] "int"

<Given> ::= "given" <IdentifierString> [<SizeMatrixExpr>]
            ":" ["set" "of"] "int" ["{" <UnboundedSetOfRanges> "}" | <AtomicId>]

<Letting> ::= "letting" <RangeVariableId> "be" (<ArithExpr> | <SetExpr>)

<Find> ::= "find" <RangeVariableId>
           ":" ["set" "of"] "int" <BoundedConstantSet>

<Minimizing> ::= ("minimizing" | "minimising") <AtomicId>
<Maximizing> ::= ("maximizing" | "maximising") <AtomicId>

<BoundedConstantSet> ::= ( "{" <BoundedSetOfRanges> "}" | <AtomicId> )

<LetFindComprehension> ::= "[" (<Letting> | <Find>) "|" <CompParamsAndConditions> "]"
```

```
<CtComprehension> ::= "[" <ConstraintExpression> "|" <CompParamsAndConditions> "]"
<SetComprehension> ::= "{" <ArithExpr> "|" <CompParamsAndConditions> "}"

<VarComprehension> ::= "[" <Id> "|" <ComprehensionParameters> "]"

<CompParamsAndConditions> ::= <RangeList> ["," <ConditionList>] | <ConditionList>

<RangeList> ::= <ParameterRange> ["," <RangeList>]
<ParameterRange> ::= <IdentifierString> "in" <BoundedConstantSet>
<ConditionList> ::= <ComprehensionCondition> ["," <ConditionList>]
<ComprehensionCondition> ::= <ArithExpr> ("!=" | "<" | ">" | "<=" | ">=" | "==")
                             <ArithExpr>

<Constraints> ::= "such that" (<Constraint>)*

<Constraint> ::= "cons" <IdentifierString> (<CtComprehension> | <ConstraintExpression>)
<ConstraintExpression> ::= <IdentifierString> "(" [<ConstraintArg>["," <ConstraintArg>]]

<ConstraintArg> ::= <IdList> | <ConstraintList>

<IdList> ::= (<Id> | "[" [<Id> ("," <Id>)*] "]")
<ConstraintList> ::= (<Constraint> | "[" [<Constraint> ("," <Constraint>)*] "]")

<ArithExpr> ::= "(" <ArithExpr> ")" |
                <Integer> | <AtomicId> |
                <ArithExpr> ("*" | "+" | "-" | "/") <ArithExpr> |
                "-" <ArithExpr>

<Integer> ::= "0" | <DigitNonZero> (<Integer>)*
<DigitNonZero> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<SetExpr> ::= "{" <BoundedSetOfRanges> "}" | <SetComprehension>

// The following two nonterminals are identical in BNF but have a different intention:
// the commas in the first separate ranges, in the second they separate dimensions.
<UnboundedSetOfRanges> ::= <OpenRange> ("," <OpenRange>)*
<MatrixIndexRanges> ::= <OpenRange> ("," <OpenRange>)*
<BoundedSetOfRanges> ::= <ClosedRange> ("," <ClosedRange>)*

<OpenRange>    ::= <ClosedRange> |  ".." | <ArithExpr> ".." | ".." <ArithExpr>
<ClosedRange>  ::= <ArithExpr> [".." <ArithExpr>]
```

```
<Id> ::= <AtomicId> | <MatrixExpr> | <VarComprehension> | <ArithExpr>
<AtomicId> ::= <IdentifierString> [<DerefMatrixExpr>]
<RangeVariableId> ::= <IdentifierString> [ "[" <MatrixIndexRange> "]" ]

<IdentifierString> ::= ( "a".."z" | "A".."Z" ) <AlNum>
<AlNum> ::= [("a".."z" | "A".."Z" | "0".."9" | "_") <AlNum>]

<MatrixExpr> ::= "flatten" "(" <MatrixExpr> ")" |
                 <Function> "(" <MatrixExpr> "," <ArithExpr> ")" |
                 <IdentifierString> "[" <MatrixIndexRange> "]"

<Function> ::= "mult" | "add"

// this is to dimension a matrix
<SizeMatrixExpr> ::= "[" <ArithExpr> ("," <ArithExpr>)* "]"

// specify an element of matrix
<DerefMatrixExpr> ::= "[" <ArithExpr> ("," <ArithExpr>)* "]"

<Comment> ::= "$" (<Anything>)* <Newline>
<Newline> ::= "\n" | "\r"
```