The Enfragmo system

October 2011

Contents

1 Introduction **2**
1.1 Credits . . . 2
1.2 Our Project . . . 2
1.3 The Enfragmo System, an Overview . . . 2

2 Tutorial **3**
2.1 Graph K-Colouring . . . 3
2.1.1 Example Theory File . . . 3
2.1.2 Example Instance File . . . 4
2.1.3 Invoking Enfragmo . . . 5
2.2 Hamiltonian Cycle . . . 5
2.2.1 Example Theory File . . . 5
2.2.2 Example Instance File . . . 7
2.3 Weighted Latin Square . . . 7
2.3.1 Example Theory File . . . 7
2.3.2 Example Instance File . . . 8
2.4 Bounded Spanning Tree . . . 9
2.4.1 Example Theory File . . . 9
2.4.2 Example Instance File . . . 10

3 Options **11**
3.1 Ground Solvers . . . 11
3.2 Gadgets for aggregates . . . 11
3.2.1 COUNT Gadgets . . . 11
3.2.2 MIN/MAX Gadgets . . . 11

4 Syntax for the Enfragmo System **12**
4.1 Problem Specification Grammar . . . 12
4.2 Instance Description Grammar . . . 14

1 Introduction

1.1 Credits

The project leaders are David Mitchell and Eugenia Ternovska. The primary developers of the Enfragmo system are Amir Avani, Xiongnan (Newman) Wu, and Shahab Tasharrofi. This manual was written by Xiongnan (Newman) Wu and Lucas Swanson.

1.2 Our Project

Computationally hard search and optimization problems are ubiquitous in science, engineering and business. Examples include drug design, protein folding, phylogeny reconstruction, hardware and software design, test generation and verification, planning, timetabling, scheduling and on and on. In rare cases, practical application-specific software exists, but most often development of successful methods requires hiring specialists, and often significant time and expense, to apply one or more computational approaches. Typical examples are mathematical programming (i.e., integer-linear programming, ILP), constraint logic programming (CLP), and development of custom-refined implementations of methods such as simulated annealing, branch-and-bound, reduction to SAT, etc.

One goal of the MX project is to provide another practical technology for solving these problems, but one which would require considerably less specialized expertise on the part of the user, thus making technology for solving such problems accessible to a wider variety of users. In this approach, the user gives a precise specification of their search (or optimization) problem in a declarative modelling language. A solver then takes this specification, together with an instance of the problem, and produces a solution to the problem (if there is one).

1.3 The Enfragmo System, an Overview

In our approach, search problems are formalized as model expansion, which is the logical task of expanding a given structure by new relations. Formally, the user is axiomatizing their problem, formalized as model expansion, in some extension of classical logic.

At present, our focus is on problems in the complexity class NP. For this case, the modelling language is based on classical first order logic (FO), and the default mechanism for constructing a solver is by "grounding": given a specification and instance, produce a formula of propositional logic that describes the solutions, and pass this to an engine that solves the propositional satisfiability problem (SAT solver).

2 Tutorial

Several examples will be used to introduce the basic usage of the Enfragmo system.

2.1 Graph K-Colouring

Graph colouring is a classic and well-studied NP-hard search problem. An instance consists of a graph and a number K, and a solution is a proper K-colouring of the graph. That is, we want a function mapping vertices to a set of K colours, so that no two adjacent vertices are mapped to the same colour.

We take our instance to consist of the graph plus the set of colours, so we have two sorts: vertices and colours. The axiomatization simply says that there is a binary relation Colour which must be a proper colouring of the vertices. So, in FO appropriate axioms could be:

$\forall\ v\ \exists\ c\ \ \text{Colour}(v, c)$
$\forall\ v\ \forall\ c1\ \ \text{Colour}(v, c1) \Rightarrow \neg\exists\ c2 < c1\ \ \text{Colour}(v, c2)$
$\forall\ v1\ \forall\ v2\ \forall\ c\ (\ \text{Colour}(v1, c) \wedge \text{Colour}(v2, c)\ ) \Rightarrow \neg\ \text{Edge}(v1, v2)$

Solving an instance of this problem with the Enfragmo system requires creating two files that the system will use as input: a theory file containing the *problem specification* and an instance file containing the *instance description*. The problem specification gives the type (sort) declarations, vocabularies, and axiomatization. The instance description gives the domains (i.e., a concrete set for each type or sort) and interpretations of the instance predicates.

2.1.1 Example Theory File

```
GIVEN:
    TYPES: Vtx Clr;

PREDICATES:
    Edge (Vtx, Vtx)
    Colour (Vtx, Clr);

FIND:
    Colour;

PHASE:
  SATISFYING:
    // every vertex has at least one colour
    !v:Vtx : ?c:Clr : Colour (v, c);

    // every vertex has no more than one colour
    !v:Vtx c1:Clr : ( Colour (v, c1) =>
      ~ ?c2:Clr < c1 : Colour (v, c2) );
```

```
    // any two vertices of the same colour do not share an edge
    !v1:Vtx v2:Vtx c:Clr :
    ( ( Colour (v1, c) & Colour (v2, c) ) => ~ Edge (v1, v2) );

PRINT:
     Colour;
```

graph-colour.bT

As can be seen above, theory files are divided into four parts. The "GIVEN" part defines the "TYPEs" and the relation symbols that are used to describe the problem. The Enfragmo system allows for two flavours of types: enumerable types (which are defined by the TYPES keyword.) and integer types (which are defined by the INTTYPES keyword.). Note that the list of types and the list of relation symbols are both terminated with a semi-colon. The "FIND" section declares which of the defined relations describe the solution to the problem (these are the "expansion predicate"). The "PHASE" section is divided into an optional "FIXPOINT" subsection, which will be described later, and a "SATISFYING" subsection. The "SATISFYING" subsection contains the constraints which must be satisfied by any solution to the problem. These constraints take the form of ASCII representations of FO formulas. The mapping from logical symbols to ASCII symbols is given in Table 1. The final section, "PRINT", lists the predicates you wish for Enfragmo to include in its output. A "//" at the beginning of a line denotes that that line is a comment and will be ignored by the Enfragmo system.

Table 1: ASCII Equivalents for Logical Symbols

Logical Symbol	$\forall$	$\exists$	$\wedge$	$\vee$	$\neg$	$\Rightarrow$	$\Leftrightarrow$
ASCII Representation	!	?	&	|	~	=>	<=>

Infix binary predicates <=, =, >, and >= are all built-in. Formally, these are abbreviations. Only variables of the same type may be compared, and all order operators reflect the same order as <. The language also includes *bounded quantifiers*, x<y, x<=y, x>y, and x>=y, which are also formally abbreviations. For example, !x y<x: ϕ is the ASCII representation of $\forall$ x $\forall$ y $<$ x ϕ, which abbreviates $\forall$ x $\forall$ y (y $<$ x $\Rightarrow$ ϕ).

2.1.2 Example Instance File

```
TYPE Vtx [1..5]
TYPE Clr ['red', 'green', 'blue']

PREDICATE Edge
  (1, 2) (1, 4) (1, 5) (2, 3) (2, 5) (3, 4) (3, 5)
```

graph-colour.bI

Instance files contain the domains of types and the extensions of instance relations. Domains may be given as a range of integers (e.g., `Vtx [1..5]`) or as an enumerated list of constant symbols (e.g., `Clr ['red', 'green', 'blue']`). For a range of integers the order is numerical; for an enumerated list it is as given. Extensions of relations are given as a set of space-separated tuples, with elements of tuples being comma-separated.

2.1.3 Invoking Enfragmo

To search for a solution to this example with the Enfragmo system, navigate to the directory containing the Enfragmo system and run the following command:

```
$␣./Enfragmo␣graph-colour.bT␣graph-colour.bI
```

where "graph-colour.bT" is the name of the theory file shown above, and "graph-colour.bI" is the name of the instance file.

2.2 Hamiltonian Cycle

Hamiltonian Cycle is another "classic" NP-hard search problem. Given a graph, find a cycle in the graph that visits every vertex exactly once. Existence is NP-complete.

In our Enfragmo axiomatization we use the built-in ordering $<$ on vertices that, together with the pair (MAX, MIN), produces a cycle that visits every vertex once. Then we require that Map() is a permutation on the vertices that maps to this cycle. The solver must also construct the actual set of edges, Hc(), in agreement with this permutation.

2.2.1 Example Theory File

```
GIVEN:
    TYPES: Vtx;

PREDICATES:
    Edge (Vtx, Vtx)
    EdgeCom (Vtx, Vtx) SuccLoop (Vtx, Vtx)
    Map (Vtx, Vtx)     Hc (Vtx, Vtx);

FIND:
    Map, Hc;

PHASE:
  FIXPOINT (EdgeCom SuccLoop) :

    // define the commutative edge relation
    INFLATE { g1:Vtx g2:Vtx :
        EdgeCom (g1, g2) <=>
        ( Edge (g1, g2) | Edge (g2, g1) ) }
```

```
    // define the looping successor relation
    INFLATE { m1:Vtx m2:Vtx :
      SuccLoop (m1, m2) <=>
      ( SUCC[Vtx] (m1, m2) | ( MAX[Vtx] = m1 & MIN[Vtx] = m2 ) ) }
;

  SATISFYING:
    // map the first map vertex to the first graph vertex
    Map (MIN[Vtx], MIN[Vtx]);

    // every map vertex is mapped to exactly one graph vertex
    ! m:Vtx : ? g1:Vtx : ( Map (m, g1) &
      ! g2:Vtx : ( Map (m, g2) => g1 = g2 ) );

    // every graph vertex is mapped to by exactly one map vertex
    ! g:Vtx : ? m1:Vtx : ( Map (m1, g) &
      ! m2:Vtx : ( Map (m2, g) => m1 = m2 ) );

    // successive map vertices are only mapped to
    // graph vertices that share an edge
    ! m1:Vtx m2:Vtx g1:Vtx :
    ( ( Map (m1, g1) & SuccLoop (m1, m2) ) =>
      ? g2:Vtx : ( Map (m2, g2) & EdgeCom (g1, g2) ) );

     // every edge shared by graph vertices mapped to by
     // successive map vertices is part of the hamiltonian cycle
     ! m1:Vtx m2:Vtx g1:Vtx g2:Vtx :
     ( ( ( Map (m1, g1) & Map (m2, g2) ) & SuccLoop (m1, m2) ) =>
     Hc (g1, g2) );


     // every edge in the hamiltonian cycle is shared by
     // graph vertices mapped to by successive map vertices
     ! g1:Vtx g2:Vtx : ( Hc (g1, g2) =>
     ? m1:Vtx m2:Vtx :
     ( ( Map (m1, g1) & Map (m2, g2) ) & SuccLoop (m1, m2) ) );

PRINT:
     Hc;
```

ham-cycle.bT

This example illustrates the use of the bi-conditional to define abbreviations (e.g. "`!g1:Vtx: !g2:Vtx<g1: (EdgeCom(g1,g2) <=> (Edge(g1,g2) | Edge(g2,g1)))`"). The predicates `EdgeCom` and `SuccLoop` are not strictly necessary to find a solution to the problem, but they do clarify the axiomatization somewhat. Also shown in this example are the order-relevant built-in functions: `MIN[type]` and `MAX[type]` and the built-in `SUCC[type](x, y)` binary relation. `MIN[type]` and `MAX[type]` refer respectively to the least and greatest elements in the domain of the specified type. `SUCC[type](x, y)` is true if and only if the

element x is succeeded by the element y in the domain of the specified type, as reflected by the order of <.

2.2.2 Example Instance File

```
TYPE Vtx [1..5]

PREDICATE Edge
  (1, 2) (1, 3) (1, 5) (2, 4) (2, 5) (3, 4) (3, 5)
```

ham-cycle.bI

2.3 Weighted Latin Square

A Latin Square (or Quasigroup) is an n by n matrix with elements in $\{1,\ldots,n\}$, where every row and every column has every possible element. In the Weighted Latin Square Problem, an instance is the set $\{1,\ldots,n\}$ plus a weight for every position in the matrix and a maximum weight. The task is to construct a Latin Square with the additional constraint that for every row, the sum of the weights of each cell in that row multiplied by the element in that row is less than the given maximum weight. i.e.,

$$\forall \text{ row} : \sum_{\text{col}=1}^{n} (\text{wt}(\text{row}, \text{col}) \cdot \text{elem}(\text{row}, \text{col})) < \text{max\_weight}$$

where wt(row, col) is the weight of a position in the matrix and elem(row, col) is the element at that position.

2.3.1 Example Theory File

```
GIVEN:
    INTTYPES: Num Weight;

PREDICATES:
    cell (Num, Num, Num);

FUNCTIONS:
    weight (Num, Num):Weight
    max_weight ():Weight;


FIND:
    cell;

PHASE:
  SATISFYING:
    // every position has exactly one element
    ! row:Num col:Num : ? elem1:Num :
```

```
      ( cell(row, col, elem1) & ! elem2:Num :
        ( cell(row, col, elem2) => elem2 = elem1 ) );

    // in every row every element appears in exactly one column
    ! row:Num elem:Num : ? col1:Num :
      ( cell(row, col1, elem) & ! col2:Num :
        ( cell(row, col2, elem) => col2 = col1 ) );

    // in every column every element appears in exactly one row
    ! col:Num elem:Num : ? row1:Num :
      ( cell(row1, col, elem) & ! row2:Num :
        ( cell(row2, col, elem) => row2 = row1 ) );

    // the sum of the weight products on each line is
    // less than the maximum weight
    ! row:Num : max_weight() >
      SUM { col:Num elem:Num ;
            elem * weight (row, col);
            cell (row, col, elem) };

PRINT:
     cell;
```

wlsquare.bT

The weighted latin square example introduces another aspect of the `GIVEN` part of a theory file: function definitions. Function definitions are similar to relation definitions, except that function definitions end with a return type. For example, the line "`weight (Num, Num, Num):Weight`" defines a function that takes three `Num` arguments and returns an element of the `Weight` domain. Functions without arguments (such as "`max_weight ():Weight`") can be used to define constant functions. Arithmetic operations, which work as one would expect, are also demonstrated in this example. Only variables of integer flavoured types can be used as part of arithmetic operations. If a variable of a non-integer type needs to be used as part of arithmetic operations, it must be casted to integers by an INTEGER operator. E.g. INTEGER{x}, where x is the variable needs to be casted. This example makes use of `SUM` aggregate function to calculate the sum of multiplied weights of a given row. This function has the following syntax: "`SUM {`<vars_to_count>; <term_to_be_summed>; <condition_to_satisfy>`}`", where "<vars_to_count>" is a standard variable declaration statement, "<term_to_be_summed>" is a proper term, and "<condition_to_satisfy>" is a FO formula. In the example, `SUM {col:Num elem:Num; elem * weight (row, col); cell (row, col, elem)}` returns 0 plus the sum of all values of `elem * weight (row, col)` across all instantiations `col and elem` for which the predicate `cell (row, col, elem)` is true.

2.3.2 Example Instance File

```
TYPE Num [1..3]
TYPE Weight [1..54]

FUNCTION weight
  (1, 1: 4)
  (1, 2: 2)
  (1, 3: 6)

  (2, 1: 1)
  (2, 2: 2)
  (2, 3: 3)

  (3, 1: 2)
  (3, 2: 1)
  (3, 3: 4)

FUNCTION max_weight (:30)
```

wlsquare.bI

As can be seen above, extensions of functions are similar to those of relations, with the return value separated from the arguments by a colon symbol (`:`).

2.4 Bounded Spanning Tree

A spanning tree of a graph is a sub-graph that is a tree and visits every vertex. In this version of the problem, an instance consists of a directed graph and a bound K, and we require a directed spanning tree in which no vertex has out-degree larger than K. Existence is NP-complete.

2.4.1 Example Theory File

```
GIVEN:
    TYPES: Vtx;
    INTTYPES: Num;

PREDICATES:
    Edge (Vtx, Vtx) Map (Vtx, Vtx) Bstedge (Vtx, Vtx);

FUNCTIONS:
    K ():Num;

FIND:
    Map, Bstedge;

PHASE:
  SATISFYING:
    // every traversal vertex is mapped to
    // exactly one graph vertex
    ! t:Vtx : ? g1:Vtx : ( Map (t, g1) &
```

```
        ! g2:Vtx : ( Map (t, g2) => g2 = g1 ) );

    // every graph vertex is mapped to
    // by exactly one traversal vertex
    ! g:Vtx : ? t1:Vtx : ( Map (t1, g) &
        ! t2:Vtx : ( Map (t2, g) => t2 = t1 ) );

    // the first traversal vertex is mapped to
    // the graph vertex that is the root of the tree
    ! g1:Vtx : ( Map (MIN[Vtx], g1) =>
      ~ ? g2:Vtx : Bstedge (g2, g1) );

    // every graph vertex that is not the root
    // has at least one parent in the tree
    ! g1:Vtx : ( ~ Map (MIN[Vtx], g1) =>
        ? g2:Vtx : Bstedge (g2, g1) );

    // the traversal vertex mapped to any graph vertex comes before the
    // traversal vertices mapped to any children of that graph vertex
    ! g1:Vtx g2:Vtx : ( Bstedge (g2, g1) =>
      ! t1:Vtx : ( Map (t1, g2) =>
        ~ ? t2:Vtx <= t1 : Map (t2, g1) ) );

    // every parent graph vertex shares an edge
    // with every one of its children
    ! g1:Vtx g2:Vtx : ( Bstedge (g2, g1) => Edge (g2, g1) );

    // every graph vertex has no more than one parent
    ! g1:Vtx g2:Vtx : ( Bstedge (g2, g1) =>
      ~ ? g3:Vtx < g2 : Bstedge (g3, g1) );

    // every graph vertex has out-degree less than or equal to the bound
    ! g1:Vtx : K() >= COUNT { g2:Vtx; Bstedge (g1, g2) };

PRINT:
     Bstedge;
```

bstree.bT

This example introduces the `COUNT` aggregate function. This function has the following syntax: "`COUNT {`<vars_to_count>; <condition_to_satisfy>`}`", where "<vars_to_count>" is a standard variable declaration statement and "<condition_to_satisfy>" is a FO formula. In the example `COUNT {g2:Vtx; Bstedge (g1, g2)}` returns the number of distinct values of `g2` for which the predicate `Bstedge (g1, g2)` is true.

2.4.2 Example Instance File

```
TYPE Vtx [1..5]
TYPE Num [1..2]
```

```
PREDICATE Edge
  (1, 2) (1, 4) (1, 5) (2, 3) (2, 5) (3, 4) (3, 5)

FUNCTION K (:2)
```

bstree.bI

3 Options

3.1 Ground Solvers

Currently, we support two kinds of ground solvers. One is MiniSat SAT solver and the other one is MXC SAT solver. The default ground solver type is "InternalMXC". To select the MiniSat SAT solver, run the Enfragmo system with following command:

```
$␣./Enfragmo␣theoryfile␣instancefile␣--GroundSolverType␣InternalMiniSat
```

3.2 Gadgets for aggregates

Users can select different gadgets for different aggregate operators.

3.2.1 COUNT Gadgets

Currently, we support four COUNT gadgets. They are "DC" gadget, "DP" gadget and "SN" gadget. [1] The default COUNT gadget is the "DC" gadget. The "CountMode" option can be used to select different COUNT gadgets. E.g. to use the "SN" COUNT gadget, run the Enfragmo system with following command:

```
$␣./Enfragmo␣theoryfile␣instancefile␣--CountMode␣SN
```

When MXC ground solver is in use, users can select "CARD" mode. In this case, no gadget will be selected and the COUNT aggregates will be directly handled by the MXC SAT solver.

3.2.2 MIN/MAX Gadgets

Currently, we support two MIN/MAX gadgets. They are "DC" gadget and "DP" gadget. The default gadget for MIN and MAX aggregate is the "DP" gadget. To use the "DC" gadget, run the Enfragmo system with following command:

```
$␣./Enfragmo␣theoryfile␣instancefile␣--MinOrMaxMode␣DC
```

[1] Details of gadgets can be found in ...URL of the paper.....

4 Syntax for the Enfragmo System

4.1 Problem Specification Grammar

```
<theory_file> ::= <given_part> <find_part> <phase_part> <print_part>

<given_part> ::= GIVEN : <types_decl> ; <funcs_decl> ;
               | GIVEN : <types_decl> ; <preds_decl> ;
               | GIVEN : <types_decl> ; <preds_decl> ; <funcs_decl> ;

<types_decl> ::= TYPES : <identifier_list> ; INTTYPES : <identifier_list>
               | TYPES : <identifier_list>
               | INTTYPES : <identifier_list>

<identifier_list> ::= | <identifier_list> <identifier>

<preds_decl> ::= PREDICATES : <preds_list>

<a_pred_DCL> ::= <identifier> ( <IdentifierListSeparatedByComma> )

<preds_list> ::= | <a_pred_DCL> <preds_list>

<IdentifierListSeparatedByComma> ::= <identifier>
                                   | <IdentifierListSeparatedByComma> , <identifier>

<funcs_decl> ::= FUNCTIONS : <funcs_list>

<funcs_list> ::= | <func_DCL> <funcs_list>

<func_DCL> ::=   <identifier> ( ) : <identifier>
               | <identifier> ( <IdentifierListSeparatedByComma> ) : <identifier>

<find_part> ::= FIND : <identifier_list> ;

<phase_part> ::= <a_phase> | <a_phase>  <phase_part>

<a_phase> ::= PHASE : <fixpoint_part> <satisfying_part>

<fixpoint_part> ::= | FIXPOINT ( <identifier_list> ) : <inflation_part> ;

<inflation_part> ::= <an_inflation> | <inflation_part> <an_inflation>

<an_inflation> ::= INFLATE <inflate_description>

<an_inflate_description> ::= { <var_DCL> : <identifier> ( <IdentifierListSeparatedByComma> ) <=>
                               <FO_formula> }

<inflate_description> ::= <an_inflate_description>
                        | <inflate_description> <an_inflate_description>
```

```
<satisfying_part> ::= | SATISFYING : <satisfying_rules>

<satisfying_rules> ::= <FO_formula> ; | <satisfying_rules> <FO_formula> ;

<FO_formula> ::= ( <FO_formula> ) | <unitary_formula>
               | <FO_formula> <connective> <unitary_formula>

<unitary_formula> ::= ( <FO_formula> )
                    | <quantifier> <var_DCL> : <FO_formula>
                    | <quantifier> <a_var_DCL> <ord_operator> <term_nodes> : <unitary_formula>
                    | <unitary_formula> <binary_operator> <unitary_formula>
                    | ~ <unitary_formula>
                    | <atomic_formula>

<atomic_formula> ::= <relation_formula>
                   | SUCC [ <identifier> ] ( <term_nodes> , <term_nodes> )
                   | <ord_relation> | TRUE | FALSE

<relation_formula> ::= <identifier> ( <args> )

<ord_relation> ::= <term_nodes> <ord_operator> <term_nodes>

<min_func> ::= MIN [ <identifier> ]

<max_func> ::= MAX [ <identifier> ]

<size_func> ::= SIZE [ <identifier> ]

<abs_func> ::= ABS ( <term_nodes> )

<func_ref> ::= <identifier> ( <args> ) | <identifier> ( )

<var_DCL> ::= <a_var_DCL> | <var_DCL> <a_var_DCL>

<a_var_DCL> ::= <identifier> : <identifier>

<args> ::= <term_nodes> | <args> , <term_nodes>

<term_nodes> ::= <a_term_node> | ( <term_nodes> ) | <term_nodes> + <term_nodes>
               | <term_nodes> * <term_nodes> | <term_nodes> - <term_nodes>

<a_term_node> ::= <var_ref> | <min_func> | <max_func> | <abs_func>
                | <func_ref> | <size_func> | <aggregate> | <int_term_node>

<aggregate> ::= COUNT { <var_DCL> ; <FO_formula> }
              | MIN { <var_DCL> ; <term_nodes> ; <FO_formula> ; <term_nodes> }
              | MAX { <var_DCL> ; <term_nodes> ; <FO_formula> ; <term_nodes> }
              | SUM { <var_DCL> ; <term_nodes> ; <FO_formula> }
```

```
<var_ref> ::= <identifier>

<int_term_node> ::= <int_number> | INTEGER { <term_nodes> }

<arit_operator> ::= + | * | -

<quant_part> ::= <quantifier> <var_DCL>;

<quantifier> ::= ? | !

<binary_operator> ::= & | '|' (or)

<ord_operator> ::= < | <= | > | >= | =

<connective> ::= & | '|' (or) | => | <=>

<print_part> ::= | PRINT : <predicates>

<predicates> ::= | <predicates> <identifier>

<identifier> ::= [a-zA-Z][0-9a-zA-Z_]+
```

4.2 Instance Description Grammar

```
<instance_file> ::= <type_parts> <pred_parts> <func_parts>

<type_parts> ::= <a_type_part> | <type_parts> <a_type_part>

<a_type_part> ::= TYPE <identifier> <range>

<range> ::= [ continous_values ]

<continous_values> ::= <integer>..<integer>

<discrete_values> ::= <a_discrete_value>
                    | <discrete_values>, <a_discrete_value>

<a_discrete_value> ::= <integer> | <string>

<pred_parts> ::= | <pred_parts> <a_predicate_part>

<a_predicate_part> ::= PREDICATE <identifier> <predicate_values>

<predicate_values> ::= | <a_predicate_value>
                     | <predicate_values> <a_predicate_value>

<a_predicate_value> ::= (<discrete_values>)
```

```
<func_parts> ::= | <func_parts> <a_function_part>

<a_function_part> ::= FUNCTION <identifier> <function_values>

<function_values> ::= <a_function_value>
                    | <function_values> <a_function_value>

<a_function_value> ::= (<func_args>:<func_return_value>)

<func_args> ::= | <discrete_values>

<func_return_value> ::= <integer> | <string>

<identifier> ::= [a-zA-Z][0-9a-zA-Z_]+

<string> ::= '[0-9a-zA-Z_]+'

<integer> ::= [0-9]+
```