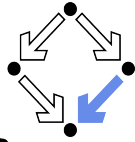


RISC

RESEARCH INSTITUTE FOR
SYMBOLIC COMPUTATION



JKU

JOHANNES KEPLER
UNIVERSITY LINZ

A DSL for Specifying a Class of Industrial Optimisation Problems

Tereso del Rio, Wolfgang Schreiner, Martina
Seidl, Temur Kutsia, Wolfgang Windsteiger

December 2025

RISC Report Series No. 25-12

ISSN: 2791-4267 (online)

Available at <https://doi.org/10.35011/risc.25-12>



This work is licensed under a CC BY 4.0 license.

Editors: RISC Faculty

B. Buchberger, R. Hemmecke, T. Kutsia, G. Landsmann, P. Paule,
V. Pillwein, N. Popov, S. Radu, J. Schicho, C. Schneider, W. Schreiner,
W. Windsteiger, F. Winkler.

Supported by: Supported by FFG project 59218671 “InProSSA: Industrial
Problem Solving Using Symbolic and Subsymbolic AI”

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

A DSL for Specifying a Class of Industrial Optimisation Problems*

Tereso del Rio^{1,2} Wolfgang Schreiner¹
Martina Seidl²
Temur Kutsia¹ Wolfgang Windsteiger¹

¹ Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

² Symbolic Artificial Intelligence Institute (SAI)
Johannes Kepler University, Linz, Austria

(Tereso.del.Rio | Wolfgang.Schreiner)@risc.jku.at
martina.seidl@jku.at
(Temur.Kutsia | Wolfgang.Windsteiger)@risc.jku.at

December 23, 2025

Abstract

This report documents a new domain-specific language (DSL) designed to express industrial optimisation problems in a Pythonic and procedural way. This language allows users to describe optimisation models using familiar programming tools such as variables, loops, and conditional statements, instead of low-level mathematical formulations.

*Supported by FFG project 59218671 “InProSSA: Industrial Problem Solving Using Symbolic and Subsymbolic AI”

Contents

1	Introduction	3
2	An Example Model	3
3	The Language	4
3.1	Types	4
3.2	Declarations	6
3.3	Statements	7
3.4	Expressions	9
4	Further Work	11
A	The Language Grammar	12
B	An Industrial Model	15
B.1	Constants	15
B.2	Objects	15
B.3	Given Objects	16
B.4	Reordering Machine	16
B.5	Cutting Machine	17
B.6	Filtering Machine	17
B.7	Check Machine	18
B.8	Pipeline	19

1 Introduction

Industrial optimisation problems often arise in domains such as logistics, manufacturing, and resource allocation, where constraints must be satisfied and objectives optimised under operational requirements. Modelling these problems typically demands a high level of expertise in constraint programming languages such as MiniZinc [5], which can form a barrier for practitioners who are more familiar with general-purpose programming languages. This is a goal shared by the project that funds this work [4].

To address this gap, we introduce a domain-specific language (DSL) designed to express industrial optimisation problems in a Pythonic and procedural way, which avoids abstractions and eases the understanding. The language offers a strongly typed but simple syntax, enabling users to specify variables, constraints, loops, assertions, and helper functions using constructs that closely resemble standard Python.

The long-term goal of this work is to support a fully automated translation pipeline from the DSL to MiniZinc. We envision a dedicated translator that will parse DSL programs, expand constructs, and ultimately produce a MiniZinc model that enables the optimisation of the problem. This approach allows domain experts to describe their optimisation problems using a Python-like specification, without needing to manually write MiniZinc code.

The rest of this report is organised as follows. In [Section 2](#), we give a high-level overview of the language by a small example model. [Section 3](#) outlines the main concepts (informal syntax) of the language and sketches their meaning (informal semantics). [Section 4](#) describes the further work planned for implementing this language. [Appendix A](#) gives a formal definition of the language's grammar; [Appendix B](#) demonstrates the language by a larger model derived from an industrial case study.

2 An Example Model

The following program describes a simple bin-packing-like problem [3]: given a number of items with a given weight and a number of boxes with a limited weight capacity, we must find an assignment of items to boxes that minimises the number of boxes used without exceeding the capacity of each box.

```
N_BOXES : int = 4
N_ITEMS : int = 5

BOX_CAPACITIES : DSLList(N_BOXES, DSInt()) = [5, 5, 5, 5]
ITEM_WEIGHTS : DSLList(N_ITEMS, DSInt()) = [4, 2, 5, 3, 1]

def not_exceed(assignments : DSLList(N_ITEMS, DSInt(1, N_BOXES))):
    """
    Checks that the assignments don't exceed the weights of the boxes
    """
    box_weight : DSLList(N_BOXES, DSInt(0, sum(ITEM_WEIGHTS)))
    for i in range(1, N_BOXES + 1):
        box_weight[i] = 0
        for j in range(1, N_ITEMS + 1):
```

```

        if assignments[j] == i:
            box_weight[i] = box_weight[i] + ITEM_WEIGHTS[j]

    assert box_weight[i] <= BOX_CAPACITIES[i]
    if box_weight[i] > 0:
        # If box i is used add 1 to the objective
        objective = objective + 1

objective : int = 0
assignments : DSLList(N_ITEMS, DSInt(1, N_BOXES))
not_exceed(assignments)

```

The program first defines the constants of this bin packing instance: the number of boxes and items, together with the capacity of each box and the weight of each item. Note that the rest of the code could be recycled for any bin packing problem.

The function `not_exceed` enforces the bin-packing constraints. It computes, for each box, the total weight of the items assigned to it using the auxiliary array `box_weight`. An `assert` then ensures that this total does not exceed the box capacity. For each box with more than 0 weight assigned, the objective counter is increased by one. The generated translation will always minimise the variable `objective` while satisfying the constraints imposed.

After specifying the instance data, the decision variable `assignments` is declared. It represents, for each item, the box to which it is assigned. This variable is then passed to `not_exceed`, which applies the capacity constraints and updates the objective.

This looks like a program that takes assignments and checks that the assertions are satisfied. But in reality, it is just imposing constraints on a free variable.

3 The Language

This language is a strongly typed subset of Python. A formal grammar of the language can be found in [Appendix A](#), but here we give an introduction to its main components.

3.1 Types

This subsection describes the primitive and composite types supported by the DSL.

Booleans. Objects of boolean type can take the values `True` or `False`. A boolean type can be described by `DSBool()` or `bool`.

Examples:

```

p : DSBool()           # p is a boolean variable.
p : bool              # p is a boolean variable.

```

Integers. Objects of integer type can take integer values and can be bounded to restrict the allowed range.

Examples:

```
n : DSInt()           # n is an unrestricted integer.
n : int              # n is an unrestricted integer.
n : DSInt(0, 10)     # n is restricted to integers between 0 and 10
                    # (inclusive).
n : DSInt(ub = 5)    # n is restricted to integers smaller than 5.
```

Floats. Objects of float type can take real values and can be bounded to restrict the allowed range.

```
x : DSFloat()        # x is an unrestricted float.
x : float            # x is an unrestricted float.
x : DSFloat(0.0, 1.0) # x is restricted to the interval [0.0, 1.0].
x : DSFloat(lb = 5.0) # x is restricted to floats greater than
                    # or equal to 5.0.
```

Lists. Objects of List type are fixed-length lists with elements of a uniform type. To access elements of the list, normal Python notation can be used* (the first index is 1).

Examples:

```
# ts is a list of 4 unrestricted integers.
ts : DSList(4, DSInt())
# ws is a list of 4 integers in the range 0..10.
ws : DSList(4, DSInt(0, 10))
# ps is a list of 3 floats greater than 3.4.
ps : DSList(3, DSFloat(3.4))

# Extracting the third element of ps
my_float = ps[3]
```

Records. An object of type record is a structured object composed of named fields with specified types. To access fields of a record, the same notation as in Python dictionaries can be used.

Examples:

```
# pos is a record with two unrestricted integer fields.
pos : DSRecord{"x": DSInt(), "y": DSInt()}

# cell is a record with a field "id" containing an int
# and a field "values" containing a list of 4 floats.
cell : DSRecord{"id": DSInt, "values": DSList(4, DSFloat())}
```

```

# Extracting the "id" of cell
cell_id = cell.id

# It is also possible to extract the second value of cell
second_value_cell = cell.values[2]

```

3.2 Declarations

This subsection introduces the different forms of declarations available in the language. It is particularly relevant to note that the type of both constants and variables has to be defined, but this task is made easier by type declarations, which allow us to abbreviate type descriptions.

Type declarations. Type declarations introduce abbreviations that simplify annotations and improve readability.

Example:

```

# abbreviation for a bounded integer type
Index = DSInt(1, 100)
# record type for a box capacity
BoxCap = DSRecord{"cap": DSInt(0, 50)}
# list of 10 bounded integers
WeightList = DSList(10, DSInt(0, 20))
# record of two integer fields
Position = DSRecord{"x": DSInt(), "y": DSInt()}
# record of two floats
FloatPair = DSRecord{"a": DSFloat(), "b": DSFloat()}

```

Constant declarations. Constant declarations introduce fixed values that parameterise the model and must include both a type and an initial value. Note that all constants must be written in capital letters.

Examples:

```

N_ITEMS : int = 5           # number of items
N_BOXES : int = 3          # number of boxes
BOX_CAP : DSList(3, int) = [10, 20, 15]
DEFAULT_POS : Position = {"x": 0, "y": 0} # default coordinate
PI_APPROX : DSFloat() = 3.14 # constant float

```

Variable declarations. Variable declarations introduce decision variables whose values will be determined by the solver, subject to the constraints of the model. Each variable name must

contain at least one lowercase letter, and there must be one declaration for each variable used.

Examples:

```
assignments : DSLList(N_ITEMS, DSInt(1, N_BOXES))
# decision variable for item to box assignment
total_weight : DSInt(0, 1000) # aggregated weight
pos : Position # using an already defined type
active : DSBool() # a boolean decision variable
weights_copy : WeightList # using another already defined type
```

3.3 Statements

Once the necessary elements in a program are declared, statements create the body of the DSL and indicate how these elements interact. The language supports the following constructs:

Assignments. Assignments introduce equality constraints between the left-hand side and the right-hand side of the expression. Note, however, that following the principle of Static Single Assignment, a new version of the variable is created each time its value is updated.

Examples:

```
x = y + 3 # constraint equating x with an expression
a[i] = values[i] * 2 # assignment to a list element
pos.x = pos.x + 1 # assignment to a field in a record
total += weight[i] # accumulated constraint
matrix[i][j] = i + j # nested list indexing
is_used = (load > 0) # boolean expression assigned as a constraint
```

Assertions. While assignments introduce equality constraints implicitly, assertions introduce explicit logical constraints that must be satisfied independently of variable definitions.

Examples:

```
assert load <= capacity # enforce constraint
assert all(weights[i] >= 0 for i in items) # ensure weights positive
assert count == sum(values) # consistency condition
assert not (x < 0) # equivalent to x >= 0
assert pos.x >= 0 and pos.y >= 0 # record-based constraint
```

Conditional statements. Conditional statements allow a model to express constraints that depend on a condition. Note that `elif` is not supported; nested `if-else` constructs must be used instead.

Examples:

```

# Simple conditional
if x > 0:
    count = count + 1

# If else structure
if load > capacity:
    violations = violations + 1
else:
    violations = violations

# Nested conditionals
if a > 0:
    if b < 3:
        total = total + 1
    else:
        total = total + 2

```

For-loops. A for-loop iterates over a finite sequence of values and will replicate its body once for each iteration during translation. Loops provide a concise way to express repeated constraints. Other Python-like iteration constructs can be used, such as `range`, `enumerate`, or direct iteration over list-typed objects. For-loops and if-statements may be freely nested.

Examples:

```

# Iteration over a bounded integer range
for i in range(1, N + 1):
    total = total + values[i]

# Enumerating over a list: idx is the index, v is the element
weights : DSLList(4, float)
weighted_avg: int = 0
for idx, v in enumerate(weights):
    weighted_avg = weighted_avg + v * idx

# Iterating directly over a list variable
weights : DSLList(4, float)
for w in weights:
    assert w >= 0

# Nested loop with a conditional
for i in range(1, N + 1):
    for j in range(1, M + 1):
        if a[i] == j:
            count = count + 1

```

Function definitions. Creates a function with or without a return statement. Note that the type of the inputs and any variable defined inside must be specified.

Example:

```
# Function without return statement
def all_below_5(xs : DSLList(n, DSInt())):
    s : DSInt(0, 1000)
    s = 0
    for x in xs:
        assert x < 5

# Function with return statement
def sum_list(xs : DSLList(n, DSInt())):
    s : DSInt(0, 1000)
    s = 0
    for x in xs:
        s = s + x
    return s
```

Return statements. A return statement provides an object existing inside a function.

Example:

```
# As in the previous example
return s
```

Function call. Invoking a function introduces all constraints defined inside it. Note that using functions as statements is only possible with those functions that do not contain a return statement

Example:

```
# Impose to my_numbers the constraints given by all_below_5
all_below_5(my_numbers)
```

3.4 Expressions

This subsection describes the expressions that may appear in the DSL. Expressions combine values, operators, and function calls to form new values.

Literals. Basic values such as integers (3), floats (2.5), booleans (True, False), and strings (if supported) can appear directly in expressions.

Arithmetic and boolean operators. Operators supported:
+, -, *, /, and, or, not, <, >, <=, >=, ==, and !=.

Examples:

```
x + 3           # arithmetic addition
x * y          # multiplication
(a > b) and flag # boolean conjunction
not my_bool    # boolean negation
count == 0     # equality comparison
```

List and record literals. Lists are written using brackets, and records use brace notation with named fields.

Examples:

```
[1, 2, 3]           # a list literal
["a", "b", "c"]    # a list of strings (if supported)
{"x": 3, "y": 7}    # a record literal with two fields
{"id": 1, "value": True} # mixed-type record literal
```

Comprehension-style any/all. Comprehension-style any (resp. all) allows us to describe a disjunction (resp. conjunction) over the elements in an object of type list.

Examples:

```
# True if some weight is greater than 5
any(w > 5 for w in weights)
# True if all weights are non-negative
all(weights[i] >= 0 for i in items)
```

Function call. A function call with a return statement can be used to substitute the objects returned.

Example:

```
def add_one(a : int):
    b = a + 1
    return b

a : int = 0
a = add_one(a) # Equivalent to a = a + 1
```

4 Further Work

The next stage of this project will develop a translator to convert programs written in the DSL into equivalent MiniZinc models. This translator will parse the typed syntax, unroll loops and conditionals, turn functions into predicates, and transform all assignments and assertions into MiniZinc constraints making use of the Static Single Assignment (SSA) [6, 2]. The translator should automatically identify the optimisation target (`objective`) and minimise this value under the constraints given. Once implemented, this translation pipeline will allow users to describe industrial optimisation problems in a familiar Pythonic style that does not require expertise in constraint programming and obtain fully executable MiniZinc models to find an optimal solution to their problems.

References

- [1] Glued laminated timber. Wikipedia, The Free Encyclopedia. Accessed: 2025-12-03.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition, 2006.
- [3] Maxence Delorme, Manuel Iori, and Silvano Martello. Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research*, 255(1):1–20, 2016.
- [4] InProSSA: Integration of Symbolic and Subsymbolic AI for Industry. RISC Software, RISC Institute, Institute for Symbolic AI, 2025. <https://www.risc-software.at/en/referenceprojects/research-project-inprossa>.
- [5] Nicholas Nethercote, Peter J. Stuckey, Guido Tack, Sebastian Brand, Gregory J. Duck, and Rémy Youssef. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming (CP 2007)*, pages 529–543. Springer, 2007.
- [6] Fabrice Rastello and Florent Bouchez Tichadou, editors. *SSA-based Compiler Design*. Springer International Publishing, 2022.

A The Language Grammar

Model: TopStatement*

TopStatement := TypeDecl | ConstDecl | FunDef | Statement

Block := Statement*

Statement := VarDecl | Stmt

Types
#####

TypeDecl := ID "=" DTypeExp

DTypeExp := DSIntType | DSFloatType | DSBoolType | DSListType | DSRecordType | ID

DSIntType := "DSInt" "(" BoundArgs ")" | "int"

DSFloatType := "DSFloat" "(" BoundArgs ")" | "float"

BoundArgs := (("lb = ")? Expr,)? (("ub = ")? Expr)?

DSBoolType := "DSBool" "(" ")" | "bool"

DSListType := "DSList" "(" ListArgs ")"

ListArgs := "length =" ConstExpr, (("elem_type = ")? DTypeExp)?

DSRecordType := "DSRecord" "(" "{" RecordField ("," RecordField)* "}" ")"

RecordField := STR ":" DTypeExp

Constants
#####

ConstDecl := UPPER_ID ":" DTypeExp "=" ConstExpr

ConstExpr := Literal | ListLiteral | RecordLiteral | ID

ListLiteral := "[" (ConstExpr,)* "]"

RecordLiteral := "{" RecordFieldLiteral ("," RecordFieldLiteral)* "}"

RecordFieldLiteral := STR ":" ConstExpr

Variables
#####

VarDecl := DTypeExpr ":" DTypeExp ("=" Expr)?

Functions definitions
#####

```

FunDef := "def" ID "(" ParamList? ")" ":" Block

ParamList := Param ("," Param)*
Param := ID ":" DTypeExp ("=" Expr)?

#####
Statements
#####

Stmt := AssignStmt
| ForStmt
| IfStmt
| ReturnStmt
| FunCall

AssignStmt := Target "=" Expr // covers x=..., a[i]=..., obj.f=...

Target := ID
| Target "." ID
| Target "[" Expr "]"

ForStmt := "for" ID "in" Iterable ":" Block
| "for" ID "," ID "in" "enumerate" "(" Iterable ")" ":" Block

Iterable := ID | ListLiteral | "range" "(" Expr "," Expr ")"

IfStmt := "if" Expr ":" Block ElseClause?

ElseClause := "else" ":" Block

ReturnStmt := "return" Expr

#####
Expressions
#####

Exps := Expr ("," Expr)*

Expr := ID
| Literal
| ListLiteral
| RecordLiteral
| FunCall
| "(" Expr ")" # to give priorities
| Expr BinOp Expr
| UnOp Expr
| AssertExpr
| AnyAllExpr

Literal := INT | FLOAT | "True" | "False"

BinOp := "+" | "-" | "*" | "/" | "%"
| "==" | "!=" | "<" | "<=" | ">" | ">="

```

```

| "and" | "or"

UnOp := "-" | "not"

AssertExpr := "assert" Expr

AnyAllExpr := AnyOrAll "(" Expr "for" ID "in" Expr ")"

AnyOrAll = "any"|"all"

#####
Functions calls
#####

Funcall := ID "(" ArgList ")"

ArgList := (Arg ("," Arg)*)?
Arg := (ID "=")? Expr

#####
Metatypes
#####

UPPER_ID: identifier in all caps (and _), used for constants.

ID: Python identifier (not all caps).

INT: integer literal.

FLOAT: floating-point literal.

STR: quoted string literal.

```



Figure 1: Pipeline overview

B An Industrial Model

Our industrial partner proposed a Glued laminated timber [1] pipeline as a case study. In this section, we produce a specification of this pipeline in the language described in this report.

The code appears here divided into sections to ease its comprehension. The code is broken into the following main parts:

- The pipeline used (see [Figure 1](#)) is described in [Subsection B.8](#).
- The machines are described in [Subsections B.4-B.5](#).
- The necessary constants and objects are described in [Subsection B.1](#) and [Subsection B.2](#).
- The initial boards to be processed by the pipeline are given in [Subsection B.3](#). Note that modifying this code would suffice to solve a similar problem for different boards (adapting constants if necessary).

B.1 Constants

```

# The following can be deduced from GIVEN_INITIAL_BOARDS
N_BOARDS : int = 1
MAX_BOARD_LENGTH : int = 30
MAX_N_INTERVALS : int = 1
# Maximum number of bad (or curved) intervals per board
MAX_N_CUTS_PER_BOARD : int = 2
# Maximum number of cuts per board (including the two fixed cuts at the start and end of the board)

N_PIECES : int = N_BOARDS * (MAX_N_CUTS_PER_BOARD - 1)
# Total number of pieces to be obtained from all boards
BEAM_LENGTH : int = 10
# Length of the beams to be produced
BEAM_DEPTH : int = 5
# Number of layers of pieces in each beam
MAX_PIECES_PER_BEAM : int = 2
# Maximum number of pieces per beam layer
MIN_DIST_BETWEEN_PIECES : int = 1
FORBIDDEN_INTERVALS : DSList(2, DSList(2, int)) = [[3,4], [7,8]]

BEAM_LENGTH : int = 10
BEAM_DEPTH : int = 3
  
```

B.2 Objects

```

Interval = DSRecord({
  "start": DSInt(0, MAX_BOARD_LENGTH),
  
```

```

    "end": DSInt(0, MAX_BOARD_LENGTH)
  })

Board = DSRecord({
  "length": DSInt(0, MAX_BOARD_LENGTH),
  "bad_intervals": DSLList(MAX_N_INTERVALS, Interval),
  "curved_intervals": DSLList(MAX_N_INTERVALS, Interval)
})

Piece = DSRecord({
  "length": DSInt(0, MAX_BOARD_LENGTH),
  "quality": DSBool()
})

CutList = DSRecord({
  "position_list": DSLList(MAX_N_CUTS_PER_BOARD, DSInt(0, MAX_BOARD_LENGTH))
})

```

B.3 Given Objects

```

GIVEN_INITIAL_BOARDS : DSLList(N_BOARDS, Board) = [
  Board(length=20,
    bad_intervals=[
      Interval(5,6),
      Interval(15,16)
    ],
    curved_intervals=[
      Interval(10,12),
      Interval(18,20)
    ]
  ),
  Board(length=15,
    bad_intervals=[Interval(3,4), Interval(18,20)],
    curved_intervals=[Interval(7,9), Interval(18,20)],
  ),
  Board(length=25,
    bad_intervals=[Interval(8,10), Interval(18,20)],
    curved_intervals=[Interval(5,7), Interval(12,14)]
  )
]

```

B.4 Reordering Machine

```

def reordering_piece_machine(list_to_reorder: DSLList(N_PIECES, Piece),
                             swapping_decisions: DSLList(N_PIECES - 1, bool),
                             ) -> DSLList(N_PIECES, Piece):

  new_list : DSLList(N_PIECES, Piece) = list_to_reorder
  for i in range(N_PIECES - 1): # Number of swapping decisions
    if swapping_decisions[i]:
      aux_piece = new_list[i]
      new_list[i] = new_list[i + 1]
      new_list[i + 1] = aux_piece

```

```
return new_list
```

```
def reordering_board_machine(list_to_reorder: DSLList(N_BOARDS, Board),
                             swapping_decisions: DSLList(N_BOARDS - 1, bool),
                             ) -> DSLList(N_BOARDS, Board):
    new_list : DSLList(N_BOARDS, Board) = list_to_reorder
    for i in range(N_BOARDS - 1): # Number of swapping decisions
        if swapping_decisions[i]:
            aux_board = new_list[i]
            new_list[i] = new_list[i + 1]
            new_list[i + 1] = aux_board
    return new_list
```

B.5 Cutting Machine

```
def cutting_machine(board_list: DSLList(N_BOARDS, Board),
                    cuts_list_list: DSLList(N_BOARDS, CutList)):
    pieces: DSLList(N_PIECES, Piece)
    for board_index, board in enumerate(board_list):
        cut_list : DSLList(MAX_N_CUTS_PER_BOARD, DSInt(0, MAX_BOARD_LENGTH)) = cuts_list_list[board_index].p
        assert cut_list[1] == 0
        assert cut_list[MAX_N_CUTS_PER_BOARD] == board.length
        curved_intervals_board : DSLList(MAX_N_INTERVALS, Interval) = board.curved_intervals
        for interval in curved_intervals_board:
            assert any(interval.start <= cut_list[cut] and cut_list[cut] <= interval.end for cut in range(1,
        for cut_index in range(2, MAX_N_CUTS_PER_BOARD):
            # Impose ordered cuts
            piece_length = cut_list[cut_index - 1] - cut_list[cut_index]
            assert piece_length >= 0
            if piece_length == 0:
                assert cut_list[cut_index] == board.length
                # No piece of length 0 unless it is the last piece
                # This reduces the spaces of solutions
                # But also removes valid solutions if there is more than one reordering machine (unless the
            bad_intervals_board : DSLList(MAX_N_INTERVALS, Interval) = board.bad_intervals
            if all(
                interval.start <= cut_list[cut_index] and
                cut_list[cut_index - 1] <= interval.end
                for interval in bad_intervals_board
            ):
                quality = 1
            else:
                quality = 0
            pieces[(board_index - 1) * (MAX_N_CUTS_PER_BOARD - 1) + cut_index] = {"quality": quality, "length":
    return pieces
```

B.6 Filtering Machine

```
def filtering_machine(list_to_filter: DSLList(N_PIECES, Piece),
                      keep_decisions: DSLList(N_PIECES, bool),
```

```

    ):
# Initialize filtered list with empty pieces
filtered_list : DSLList(N_PIECES, Piece)
for i in range(N_PIECES):
    if keep_decisions[i]:
        assert list_to_filter[i].quality == 1
        filtered_list[i] = list_to_filter[i]
    else:
        objective += list_to_filter[i].length
        filtered_list[i] = {"quality": 1, "length": 0}

return filtered_list

```

B.7 Check Machine

```

def checking_machine(pieces: DSLList(MAX_PIECES_PER_BEAM, elem_type = Piece)):
    depth = 0
    length = 0
    n_length = 0
    n_prev_layer = 0
    new_beam = 1
    current_index = 0
    for piece in pieces:
        current_index = current_index + 1
        length = length + piece.length
        n_length = n_length + 1
        assert length <= BEAM_LENGTH
        if length == BEAM_LENGTH:
            depth = depth + 1
            n_prev_layer = n_length
            n_length = 0
            length = 0
            if depth == BEAM_DEPTH:
                new_beam = 1
                depth = 0
        else:
            new_beam = 0
            for i in range(1, MAX_PIECES_PER_BEAM):
                if i < n_prev_layer:
                    start = current_index - n_length - n_prev_layer + 1
                    end = start + i - 1
                    s = 0
                    for j in range(1, MAX_PIECES_PER_BEAM):
                        # the language can be made more elegant taking in for from i to j, and automatically
                        if start <= j:
                            if j <= end:
                                s = s + pieces[j].length
                                assert abs(s - length) >= MIN_DIST_BETWEEN_PIECES
    # Check forbidden intervals
    for interval in FORBIDDEN_INTERVALS:
        assert not (interval[1] <= length and length <= interval[2])

```

B.8 Pipeline

```
initial_boards: DSLList(N_BOARDS, Board) = GIVEN_INITIAL_BOARDS

# Swapping boards

swapping_decisions_boards: DSLList(N_BOARDS - 1, bool)
swapped_boards: DSLList(N_BOARDS, Board) = GIVEN_INITIAL_BOARDS
swapped_boards = reordering_board_machine(initial_boards, swapping_decisions_boards) # It's necessary to sp

# Cutting boards

cuts_list_list: DSLList(N_BOARDS, CutList)
pieces : DSLList(N_PIECES, Piece)
pieces = cutting_machine(initial_boards, cuts_list_list)

# Filtering pieces

keep_decisions: DSLList(N_PIECES, bool)
filtered_pieces: DSLList(N_PIECES, Piece)
filtered_pieces = filtering_machine(pieces, keep_decisions)

# # Reordering pieces

swapping_decisions: DSLList(N_PIECES - 1, bool)
reordered_pieces: DSLList(N_PIECES, Piece)
reordered_pieces = reordering_piece_machine(filtered_pieces, swapping_decisions)

# # Checking pieces

checking_machine(reordered_pieces)
```