CP Report

p.soleimanybaraijany@studio.unibo.it[1],
maryam.salehi@studio.unibo.it[2], and
shadi.farzankia@studio.unibo.it[3]

[1]University of Bologna

September 2022

1 Introduction

Very-large-scale integration(VLSI) is the process of embedding numerous transistors onto a silicon semiconductor microchip. VLSI technology emerged in the late 1970s when microcomputer chips were developing. The last few decades have witnessed remarkable growth in the electronics industry thanks to the advancements in VLSI technology. Presently, cellular communication and smartphones afford unprecedented processing capabilities and portability due to technological improvements and it is forecasted that this trend will continue as there is an ever-increasing demand for state-of-the-art devices. In VLSI design, maintaining a small area and high performance has always been two conflicting constraints considered by integrated circuit designers. In this project, we try to propose a solution to the size problem of VLSI circuits. A certain number of circuits are embedded in a plate with a fixed width and the ultimate goal is to minimize the height of the plate. Four different approaches including constraint programming (CP), Propositional Satisfiability (SAT), Satisfiability Modulo Theories (SMT), and linear programming (LP), have been utilized to solve the problem and the results of all approaches have been compared and analyzed to find the best solution. This report explains how Constraint Programming was used to solve the VLSI problem[1].

2 Instance Format and Solution Format

In this section, the format of input VLSI instances as well as the output solutions are presented.

2.1 Instance Format

Input instances are defined as follows:

w
n
$x_0 \quad y_0$
$x_1 \quad y_1$
...

Where:

w: width of the plate
n: number of circuits to be embedded
x_i: the horizontal dimension of the ith circuit
y_i: the vertical dimension of the ith circuit

For example, a file with the following lines:

9
5
3 3
2 4
2 8
3 9
4 12

describes an instance in which the silicon plate's width is 9, and we need to place 5 circuits, with the dimensions 3×3, 2×4, 2×8, 3×9, and 4×12. Figure 1 shows the graphical representation of the instance.

2.2 Output format

After solving the problem using the input instances, the output can be represented as:
$w \quad max\_height$
n
$x_0 \quad y_0 \quad pos_{x_0} \quad pos_{y_0}$
$x_1 \quad y_1 \quad pos_{x_1} \quad pos_{y_1}$
...

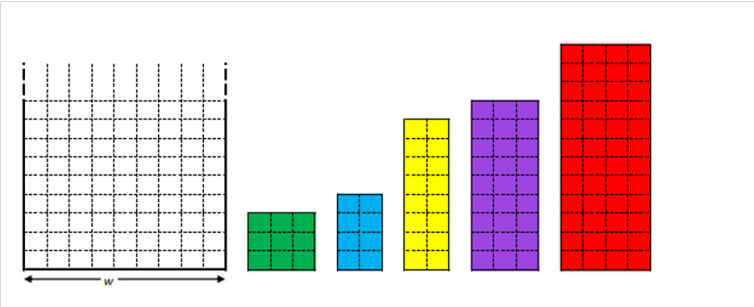


Figure 1: Graphical representation of an instance

where:

w, n, xi, yi : same as input instance
max_height: The optimal height of the plate
pos_{x_i}= horizontal coordinate of the bottom left point of the ith circuit
pos_{y_i}= vertical coordinate of the bottom left point of the ith circuit
For example, a possible solution for the instance presented in figure 1 can be as follows:

```
9    12
5
3    3    4    0
2    4    7    0
2    8    7    4
3    9    4    3
4    12    0    0
```

Which states that the maximum height is 12 and the left bottom corner of the 3×3 circuit is at (4,0). The output has been depicted in figure 2.

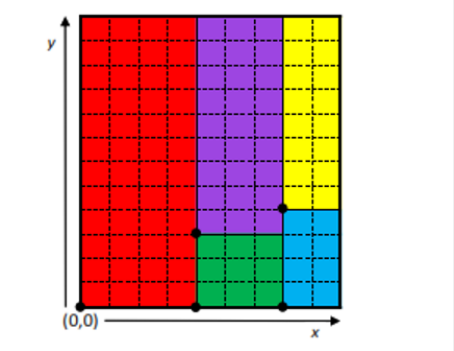


Figure 2: Graphical representation of one solution

3 Preliminary Concepts and Modeling

Initially, Constraint Programming (CP) was used to solve the problem. The first part of modeling included model selection and defining the proper variables.
The main concept of modeling is that this problem can be considered a task scheduling problem and hence it is possible to use the cumulative constraint. the problem of putting rectangles(circuits) on a plate can be considered a task scheduling problem. Hence, it is possible to use the cumulative constraint. Two different views of the problem at hand can be assumed. If we consider circuits as activities in the task scheduling problem, from one point of view, duration is the vertical lengths of the circuits(y_i), the amount of resource each activity needs is the horizontal lengths of the circuits(x_i), and the total number of resources is equal to the width of the plate (w).
From another point of view, duration is the horizontal length of the circuits(x_i), the amount of resources needed by each activity is the vertical length of the circuits(y_i), and the total number of resources is equal to the height of the plate.

3.1 Variables and Domains

3.1.1 Maximum and Minimum Height

The width of the plate is fixed and is equal to w, However, the height of the plate is a variable that should satisfy two criteria:

1. The height should be determined such that the area of the plate is minimized.

2. We should be able to embed all the circuits on the plate without overlapping.

Considering the two criteria above, the plate's minimum height is the maximum of the maximum width of the circuits embedded on the plate and the sum of the areas of all the circuits divided by the width of the plate.

$$\text{min\_height} = \max(\max(y), \text{ceil}(\text{sum}([y[i]*x[i] \mid i \text{ in } 1..n])/w)$$

The height of the plate cannot be less than the maximum of the widths of the circuits embedded on the plate. This is because in the best case, all circuits can be placed in only one row. However, in most cases, we need to have more than one row and a good estimation of the minimum height can be achieved by dividing the sum of the areas of all the circuits by the width of the plate which is fixed.
In order to determine the maximum height, one needs to consider the worst case in which all of the circuits have the maximum width and only one circuit can be embedded in each row. As a result, the maximum height is equal to the sum of the maximum width of the circuits.

$$\text{max\_height} = \text{sum}([y[i] \mid i \text{ in } 1..n])$$

3.1.2 Position of the Circuits

To determine where each circuit is located on the plate, only the coordinates of its left-bottom corner are kept. Variables pos_x and pos_y denote the horizontal and vertical coordinates of the left-bottom corner of the circuits respectively.
pos_x varies between zero and width minus $min(x)$, where $min(x)$ is the minimum of the lengths of the circuits.
pos_y varies between zero and maximum height minus min(y), where min(y) is the minimum of the widths of the circuits.

array [1..n] of var 0..w - min(x): pos_x
array [1..n] of var 0..max_height - min(y): pos_y

4 Constraints

4.1 Implied and Main Constraints

Implied constraints (also called redundant in some solvers) are the constraints that improve the propagation and thus reduce the time needed to perform the search.
While implied constraints do not result in a different answer, they are useful since the time for solving the problem can be decreased significantly.
Three $forall$ constraints were used to indicate the circuits' coordinates, two of which determined the position of the left-bottom corner of circuits[2].

constraint forall(i in 1..n) ($pos_x[i]$ >= 0 /\ pos_x[i] <= w - min(x))

constraint forall(i in 1..n) (pos_y[i] >= 0 /\ pos_y[i] <= height-min(y))

The third forall constraint has been put to ensure that circuits will be embedded within the boundaries of the plate.

constraint forall(i in 1..n) (pos_x[i] + x[i] <= w /\ pos_y[i] + y[i] <= height)

Since we had to put several circuits on the plate, we had to make sure that these circuits do not overlap. Constraint diffn was used to ensure that rectangular circuits are not overlapping.

constraint diffn(pos_x, pos_y, x, y)

As was stated in the modeling section, the problem of putting rectangles(circuits) on a plate can be considered a task scheduling problem. Hence, it is possible to use the cumulative constraint. Two different views of the problem at hand can be assumed. If we consider circuits as activities in the task scheduling problem, from one point of view, duration is the vertical lengths of the circuits(y_i), the amount of resource each activity needs is the horizontal lengths

of the circuits(x_i), and the total number of resources is equal to the width of the plate (w).
From another point of view, duration is the horizontal length of the circuits(x_i), the amount of resources needed by each activity is the vertical length of the circuits(y_i), and the total number of resources is equal to the height of the plate.

constraint cumulative(pos_y, y, x, w)
constraint cumulative(pos_x, x, y, max_height)

4.1.1 Symmetry Breaking Constraints

To avoid repetitive solutions, symmetry-breaking constraints are necessary. We may face horizontal, vertical, or 180 degrees symmetry. Two constraints were added to break these symmetries.
The first constraint is concerned with putting the circuit with the biggest area at position (0,0). To achieve this goal circuits are ordered at the beginning of the solution. The circuit corresponding to index one is the one having the biggest area.

constraint $pos_x[1] == 0 \ /\backslash \ pos_y[1] == 0$

The second symmetry-breaking constraint is a forall constraint in conjunction with a lex-less constraint. By using these constraints, when two circuits have the same size, the one with the lower index will be chosen.

constraint forall(i in 1..n-1, j in 2..n where i¡j) (if (x[i]==x[j] /\ y[i]==y[j])

then lex_less([pos_x[i], pos_y[i]], [pos_x[j], pos_y[j]]) endif)

5 Search

5.1 Annotations

In MiniZinc, no declaration of how solutions should be searched exists. Sometimes we need to determine how the search should be done particularly in case of combinatorial problems but the search strategy is not a part of the model. The implementation of all the available search strategies is not needed and annotations are used to communicate the extra information to the constraint solver. To choose and constrain variables various annotations are available, some of which were used in this project. The results of using different variables were compared to determine the best combination.
Variable choice annotations:

- input_order: choose in order from the array
- first_fail: choose the variable with the smallest domain size

- smallest: choose the variable with the smallest value in its domain
- dom_w_deg: choose the variable with the smallest value of domain size divided by weighted degree, which is the number of times it has been in a constraint that caused failure earlier in the search

Constraining variable annotations:

- indomain_min: assign the variable its smallest domain value
- indomain_median: assign the variable its median domain value (or the smaller of the two middle values in case of an even number of elements in the domain)
- indomain_random: assign the variable a random value from its domain
- indomain_split bisect the variables domain excluding the upper half

5.2 Restart

Using depth-first search to solve optimization problems may result in wrong decisions at top of the search tree that can take a long time to undo. To mitigate this problem, we can restart the search from the top which may result in different decisions.
MiniZinc includes annotations to control restarts that are attached to the solve item of the model. Various restart annotations can be used to control the frequency of restart occurrence. Restarts happen when we reach a limit in nodes. When a restart happens, the search will start from the top of the tree again.

- restart_constant($< scale >$) where ¡scale¿ is an integer defining after how many nodes to restart.
- restart_linear($< scale >$) where ¡scale¿ is an integer defining the initial number of nodes before the first restart. The second restart gets twice as many nodes, the third gets three times, etc.
- restart_geometric($< base >$, $< scale >$) where $< base >$ is a float and $< scale >$ is an integer. The kth restart has a node limit of $< scale >$* $< base >^k$.
- restart_luby($< scale >$) where ¡scale¿ is an integer. The kth restart gets $< scale >$*L[k] where: mzn‘L[k]‘ is the kth number in the Luby sequence. The Luby sequence looks like 1 1 2 1 1 2 4 1 1 2 1 1 2 4 8 ..., that is it repeats two copies of the sequence ending in 2^i before adding the number 2^{i+1}.

5.3 Free Search

Adding annotations to the model is beneficial but to take full advantage of chuffed's performance, chuffed should be allowed to switch between its activity-based search and defined search. To enable this feature, free search should be added in the command line option or free search should be activated in the solver configuration pane of the MiniZinc IDE. In this project, we use the 'free_search' argument in the 'solve' method of the Minizinc library in python. By using free search, the solver may ignore any search annotations but is not required to do so. After using free search the speed of solving the problem increased dramatically. We show the effect of activating free search on solving time in the result section.

6 Rotation

The second variant of the VLSI problem concerns the case in which rotation is admissible. Hence, each circuit can be rotated by 90 degrees and its width and height can be swapped. In order to take into consideration, the rotation we introduced three new variables $actual_x$, $actual_y$, and rot.
Variable rot is a Boolean array with a size equal to the number of circuits:

array[1..n] of var 0..1: rot

Variable $actual_x$ and $actual_y$ are the representation of x array and y array respectively, considering the rotation.

array[1..n] of var 0..max(w, sum(y)): $actual_x$ = [x[i]*(1-rot[i])+y[i]*rot[i] | i in 1..n]

array[1..n] of var 0..max(w, sum(y)): $actual_y$ = [y[i]*(1-rot[i])+x[i]*rot[i] | i in 1..n]

The solver decides whether each element of the rotation array should be filled with zero or one. If the solver chooses zero in the rotation array for one circuit, it means the height and width of the circuit will not be swapped and the $actual_x$ and $actual_y$ of that circuit are the same as x and y respectively.
If the solver chooses one in the rotation array for one circuit, it means the height and width of the circuit will be swapped and the $actual_x$ and $actual_y$ of that circuit are y and x respectively.

6.1 Symmetry Breaking and Rotation

All the constraints needed to solve the VLSI problem with rotation are the same as the variant without rotation. The only difference is that instead of x and y, we have used $actual_x$ and $actual_y$ to consider whether weather rotation has occurred or not.

The only new constraint that has been added to the VLSI problem with rotation is a forall constraint to ensure rotation occurs when it is possible and necessary:

1. If the width and height of a circuit are the same, rotation is not necessary.
2. If the height of a circuit is bigger than w (width of the plate), rotation is impossible.

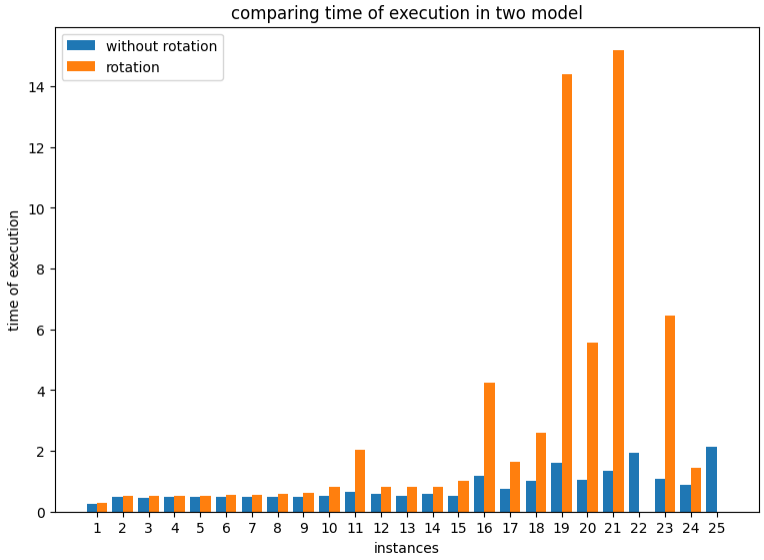
   constraint forall(i in 1..n) (if x[i]==y[i] \/ y[i] > w then rot[i] = 0

7 results

We used the Chuffed solver as it performed better than the other solvers like Gecode. After adding all the mentioned constraints in the previous sections, we still did not acquire acceptable results. Therefore, we decided to work on our solver by changing its configuration. By activating free search, we observed a remarkable improvement in the number of solved instances. Then we ran several possible combinations of the search heuristics and restart strategies.
The experimental results showed that the following combination has the best performance with respect to the number of solved instances as well as the time of execution:

first_fail, indomain_median, linear

Free search	Variable Search Heuristics	Search domain Heuristics	Solved instances	restart	Time (m)	Avg time
False	dom_w_deg	indomain_min	28	Linear	4350.74	24.28
True	**dom_w_deg**	**indomain_min**	**39**	No restart	**787.80**	**4.43**
True	dom_w_deg	indomain_min	37	luby	1586.52	9.86
True	dom_w_deg	indomain_median	39	No restart	858.02	6.28
True	dom_w_deg	indomain_split	37	No restart	1387.48	4.60
True	dom_w_deg	indomain_split	35	linear	1672.78	4.52
True	Input_order	indomain_median	38	No restart	1054.58	11.1
True	Input_order	indomain_median	37	constant	1145.92	6.22
True	Input_order	indomain_median	37	luby	1354.88	11.56
True	Input_order	indomain_median	34	none	2209.93	11.56
True	Input_order	indomain_min	36	none	1615.75	10.93
True	Input_order	indomain_min	36	luby	1573.95	9.50
True	**first_fail**	**indomain_median**	**39**	**linear**	**663.54**	**8.65**
True	first_fail	indomain_median	37	No restart	1204.45	7.68
True	first_fail	indomain_median	37	constant	1581.26	17.72
True	first_fail	indomain_min	36	constant	1426.60	5.89
True	first_fail	indomain_median	35	none	1939.66	12.11
True	first_fail	indomain_split	38	No restart	1242.35	8.74
True	first_fail	indomain_splt	36	linear	1440.60	5.93



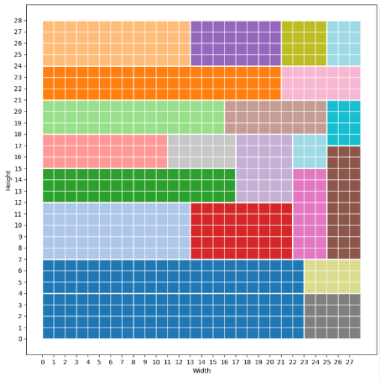


Figure 3: with Rotation

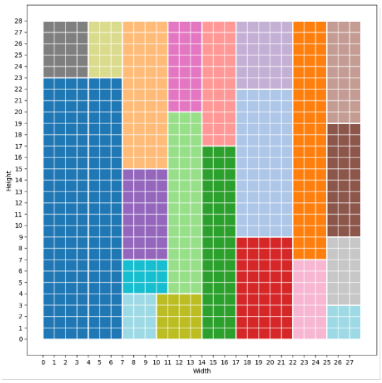


Figure 4: without Rotation

8 Example

You can see the solution for instance 21 without rotation and with rotation above.

References

[1] Mikael Östlund. "Implementation and Evaluation of a Sweep-Based Propagator for Diffn in Gecode,Thesis, Uppsala universi". In: (2017). DOI: 10.1109/ISED.2016.7977061.

[2] *The MiniZinc handbook*. 2018. URL: https://www.minizinc.org/doc-2.5.5/en/.