VLSI Report

Using CP & SMT models to solve VLSI problems of increasing difficulty

Alessandro Maggio[1] and Serban Cristian Tudosie[2]

[1]alessandro.maggio5@studio.unibo.it
[2]serban.tudosie@studio.unibo.it

August 2021

Abstract– We aim at comparing Constraint Programming (CP) and Satisfiability Modulo Theories (SMT) techniques in order to solve problems of Very Large Scale Integration (VLSI). We have built two models: one using MiniZinc with the standard CP theory while the other one employes a similar problem model expressed in First Order Logic with new specific contraints. At the end of this report we are proud to expose an original way which may be very effective to implement a SAT version of VLSI problems.

1 THE VLSI PROBLEM

Very Large Scale Integration (VLSI) is a well known problem since the modern digital electronic was born. The core problem consists in finding the best disposition of chips on a circuit plate in order to minimize the overall size of the device. In our specific case, we have a fixed plate width for every problem instance and all the chips that must be included respecting the plates' size constraints. Then the height of the plate must be minimize.

Our work is entirely available on GitHub.
The CP approach, obtained with MiniZinc, and the SMT approach made in Python, are perfectly operative. There is also an alternative proposal for an original SAT formulation.

2 CONSTRAINT PROGRAMMING

The most intuitive way to solve VLSI problem is using CP techniques. For this purpose we have used MiniZinc and several libraries and solver it has available, trying to achieve a good enough problem modelling to find a solution for all the 40 instances.

2.1 Problem formulation

We noticed that the task has a strong analogy with the scheduling problem. Since we have to find positions for the given chips, we decided to use the global `cumulative` and `diffn` constraints of MiniZinc together with standard domain constraints. The following lexical conventions and equations show a rigorous way to explain our formulation.

Parameters:

- n number of chips
- w maximum plate width
- $chips\_w_i$ width of the i_th chip
- $chips\_h_i$ height of the i_th chip

Variables:

- h plate's height that must be minimized
- x_i bottom left x coordinate of the i_th chip
- y_i bottom left y coordinate of the i_th chip

The coordinates, after the assignment, represent a complete solution of a problem instance. The height variable is crucial to evaluate the solution quality.

Constraints:

$$0 \leq y_i \leq h - chips\_h_i \qquad i \in [1, n] \tag{1}$$

$$0 \leq x_i \leq w - chips\_w_i \qquad i \in [1, n] \tag{2}$$

$$\forall i, j.$$
$$y_i + chips\_h_i \leq y_j \bigvee y_j + chips\_h_j \leq y_i \tag{3}$$
$$\bigvee x_i + chips\_w_i \leq x_j \bigvee x_j + chips\_w_j \leq x_i$$

$$\forall j \sum_{i \mid y_i \leq j \leq y_i + chips\_h_i} chips\_w_i \leq w \qquad j \in [0, h) \tag{4}$$

$$min\_h \leq h \leq max\_h \tag{5}$$

2.2 Constraints Implementation

The first two constraints (1)(2) reduce the domain of the chips positions to only the places which avoid to make the chips exceed the plate's boundaries. Since in MiniZinc an array is declared such that all its elements must have the same domain range, it is not possible to operate the exact initialization expressed previously. However a quite accurate domain definitions are $0 \leq y_i \leq h - min(chips\_h)$ and $0 \leq x_i \leq w - min(chips\_w)$ combined with two extra constraints that emulate the desired behaviour (Code 1).

```
set of int: CHIPS = 1..n;
array[CHIPS] of var 0..w - min(chips_w): x_positions;
array[CHIPS] of var 0..max_h - min(chips_h): y_positions;
constraint max(i in CHIPS)(chips_h[i] + y_positions[i]) <= h;
constraint max(i in CHIPS)(chips_w[i] + x_positions[i]) <= w;
```

Code 1: Max Constraints

The third constraint (3) forces chips not to overlap. In order to achieve it, we preferred a MiniZinc library global constraint called *diffn*; in our case it can be formulate, according to our parameters and variables names, as

$$\texttt{diffn}(\{x_1 ... x_n\}, \{y_1 ... y_n\}, \{chips\_w_1 ... chips\_w_n\}, \{chips\_h_1 ... chips\_h_n\})$$

The cumulative constraint (4) is used along the vertical direction, exploiting a scheduler-like point of view, in our case formulated, still according to our parameters and variables names, as

$$\texttt{cumulative}(\{y_1...y_n\}, \{chips\_h_1...chips\_h_n\}, \{chips\_w_1...chips\_w_n\}, w)$$

MiniZinc offers an internal and efficient implementation of it and returns the y array. We tried to add a very similar constraint for the horizontal direction:

$$\texttt{cumulative}(\{x_1...x_n\}, \{chips\_w_1...chips\_w_n\}, \{chips\_h_1...chips\_h_n\}, h)$$

Sadly it reduces the performance, forcing to omit this second constraint. The last constraint (5) is useful in order to reduce the search space for h, providing to the solver the smallest possible range. The maximum possible value happens when all the chips are on a single column, so $max\_h = sum(chips_h)$. The minimum value is obtained when there are no empty spaces among the chips and all the plate width is fitted; mathematically it can be written as:

$$min\_h = \frac{\sum_{i \in [1,n]} chips\_w_i * chips\_h_i}{w}$$

2.3 Symmetry breaking constraint

There are two main axis of symmetry parallel to the plate's borders. Mirroring the disposition along the horizontal axis we encountered a problem: MiniZinc cannot iterate over h since it is not grounded yet and using max_h would be very computationally expensive since it includes a lot of empty space. Then we proceeded to break the symmetry along the vertical axis as shown in (Code 2) but it appears a critical disadvantage. Only few instances benefit from its introduction while most cases are solved faster without it.

```
constraint symmetry_breaking_constraint(
    lex_lesseq([x_positions[k] | k in CHIPS],
    [w - x_positions[k] - chips_w[ k] |k in CHIPS]));
```

Code 2: Symmetry breaking

Some other possible symmetries can be detected, specially in particular cases (i.e. square solutions), but often the effort to remove them is not worth.

2.4 Rotation case

Adding the possibility to rotate any chip, the problem grows significantly and it must be managed as efficiently as possible. We decided to introduce two new arrays for chips_w and chips_h in order to handle a rotation as a swap between the width and the height of a chip. The new arrays chips_w_true and chips_h_true are kept consistent with respect to the real chips' shapes using a boolean array. The constraint (Code 3) that achieves this consistency is shown in the following code fragment.

```
array[CHIPS] of var 0..max(max(chips_w), max(chips_h)):chips_w_true;
array[CHIPS] of var 0..max(max(chips_w), max(chips_h)):chips_h_true;

constraint forall(i in CHIPS)
((rotations[i] /\ chips_w_true[i] == chips_h[i] /\
chips_h_true[i] == chips_w[i])
xor (not(rotations[i]) /\ chips_w_true[i] == chips_w[i]
/\ chips_h_true[i] == chips_h[i]));
```

Code 3: Rotation

2.5 Search Strategy & Solver

The search strategy is a crucial part of Constraint Programming. Making many tries, we found that by searching first on y then on x and finally on h we reached the peak performance of this model. In order to implement it, we used search annotation as shown in (Code 4). As we can see, the solver chooses always the smallest value of the domain (`indomain_min`) in all the three cases. For the positions coordinates we used the `first_fail` strategy which imposes to pick the variable with the smallest domain. For h we noticed that there are no significant differences among several variable-selection policies because there are no points choice since `h` is a single variable.

```
      solve :: seq_search([
          int_search(y_positions, first_fail, indomain_min),
          int_search(x_positions, first_fail, indomain_min),
          int_search([h], smallest, indomain_min)])
          minimize h;
```

Code 4: Search

This construction for the solver settings let us to fully exploit the Chuffed solver capabilities. According to the documentation, the search annotations (enabled with Free Search command) results in a very good implementation. Empirically we noticed that, although Chuffed does not support effectively restart strategies yet, Gecode in every case shows longer time of computation even on simple problems. Lacking of an important resource like restarting with different seed during a single run, Chuffed precludes some interesting possible improvements.
Running different Solver and search heuristics we found that these settings shows the best performances even when rotation is allowed.

3 SATISFIABILITY MODULO THEORIES

This type of problem can be also translated efficiently in SMT. In particular we used the Z3 Python library in order to do that. We want to compare, at the end of the analysis, which method will be faster to solve the given instances. In this case is not possible to apply particular heuristics during the searching phase.

3.1 Problem formulation

The implementation of the VLSI problem is different in the case of SMT. The possibility to use Python allowed us to keep the whole solution elegantly in a class.
The parameters and the variables are the same as the model in the Constraint Programming formulation.
We have the same constraints given by the equations (1)(2)(3) for the domain and the not overlapping respectively. We do not need to add the extra constraints for the domain since we can initialize the variables respecting (1)(2). So adding two more constraints as in (Code 1) would be redundant. The difference appears in the translation of the cumulative constraint following the equation (4). This time we performed a sum over rows and a sum over columns which resulted in a faster computation by the solver (6)(7).

$$\forall u \sum_{i \mid y_i \leq u \leq y_i + chips\_h_i} chips\_w_i \leq w \qquad u \in [0, h), i \in [1, n] \tag{6}$$

$$\forall u \sum_{i \mid x_i \leq u \leq x_i + chips\_w_i} chips\_h_i \leq h \qquad u \in [0, w), i \in [1, n] \tag{7}$$

3.2 Rotation case

We follow the approach the same approach also in the case of SMT since it allows a clean re-modulation of the problem without adding redundant code. So in order to do so we added the two arrays and the following constraint to take in account rotations.

$$\begin{gathered} \forall_{i \in [1..n]} \\ (rotations_i \wedge chips\_w\_true_i = chips\_h_i \wedge chips\_h\_true_i = chips\_w_i) \\ \otimes \\ (\neg(rotations_i) \wedge chips\_w\_true_i = chips\_w_i \wedge chips\_h\_true_i = chips\_h_i) \end{gathered} \tag{8}$$

Very small code changes have been needed in order to implement this feature with Z3 thanks to the high modularity of an Object Oriented paradigm.

3.3 Optimization & Performances

In order to minimize the variable h we tried to solve the problem starting from min_h and gradually increasing h until its upper bound max_h, if a solution has not been found with the current height.
Although the expected performances were much worse than CP, mainly due to the poor search strategy customization, SMT revealed quite good overall performances on many instances.

4 PERFORMANCES ANALYSIS

We collected data from all the 40 instances both in case of chips rotation allowed and not allowed. We set the cut-off time to 300 seconds, noticing as expected several differences in CP e SMT performances. Nevertheless, the two approaches can not solve some hard instances in this short timespan therefore the rotational case is empirically always slower when also a non-rotational solution has been found. In just two cases only the availability of rotated chips made easier finding an optimal solution (probably due to a bit of luck during the search). The collected data are the following (at 3 significant digits):

	CP		SMT	
Instance	No Rotation	Rotation	No Rotation	Rotation
1	.161	.173	.009	.019
2	.163	.171	.009	.027
3	.164	.168	.014	.027
4	.170	.200	.021	.061
5	.168	.179	.041	.170
6	.168	.255	.062	.125
7	.172	.284	.059	.146
8	.180	.254	.032	.096
9	.177	.255	.087	.634
10	.200	.343	.326	1.06
11	.369	1.17	2.69	110
12	.336	1.12	.769	30.1
13	.221	.277	.379	6.99
14	.222	.306	.196	51.9
15	.228	.341	.437	12.6
16	1.17	8.85	7.70	-
17	.296	1.25	.921	-
18	.315	4.03	14.1	-
19	2.03	34.3	28.7	-
20	1.07	7.02	17.9	-
21	.718	18.4	61.7	-
22	1.57	-	241	-
23	1.26	22.3	25.9	-
24	.525	5.53	8.21	130
25	3.58	-	-	-
26	.678	6.83	34.8	-
27	.841	28.9	32.0	-
28	1.57	104	49.4	-
29	.692	13.4	28.4	-
30	29.4	-	236	-
31	.247	2.65	12.3	-
32	291	-	-	-
33	3.02	15.8	39.9	-
34	2.20	2.58	76.5	-
35	1.06	1.20	35.8	-
36	.451	.839	-	48.1
37	102	-	-	-
38	-	-	-	-
39	.669	19.7	-	-
40	-	-	-	-

Table 1: Solving Time Results

We created also a more user-friendly way to visualize the data, which highlights the general trend and the best results obtained in the non rotational-case.

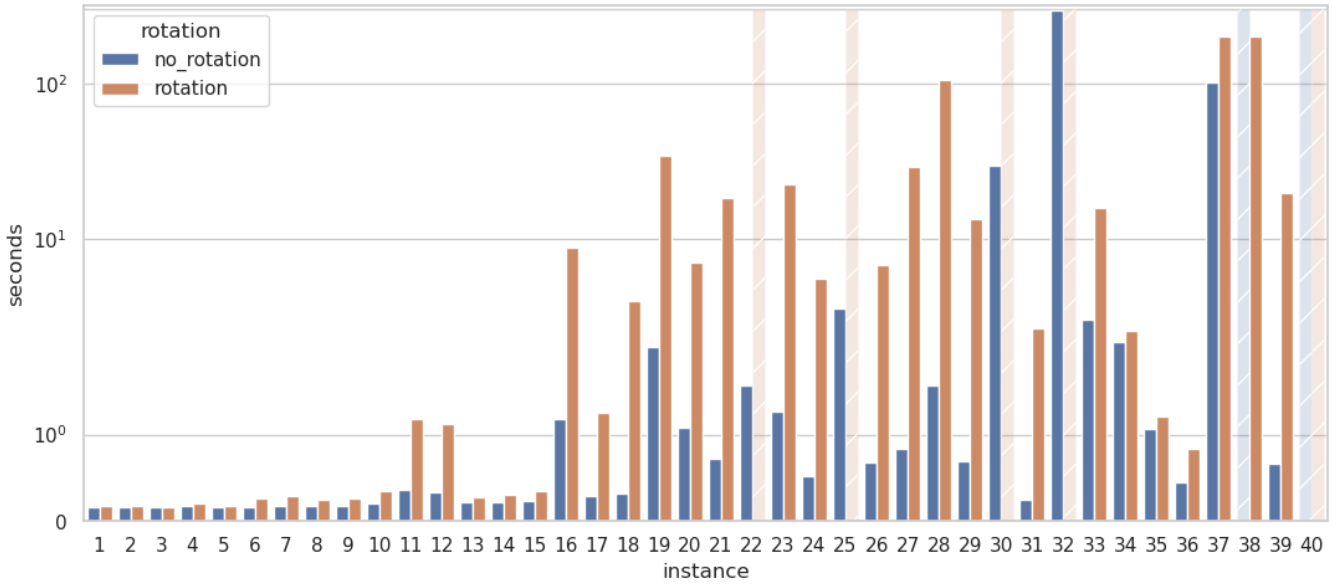

Figure 5: CP Solving Times

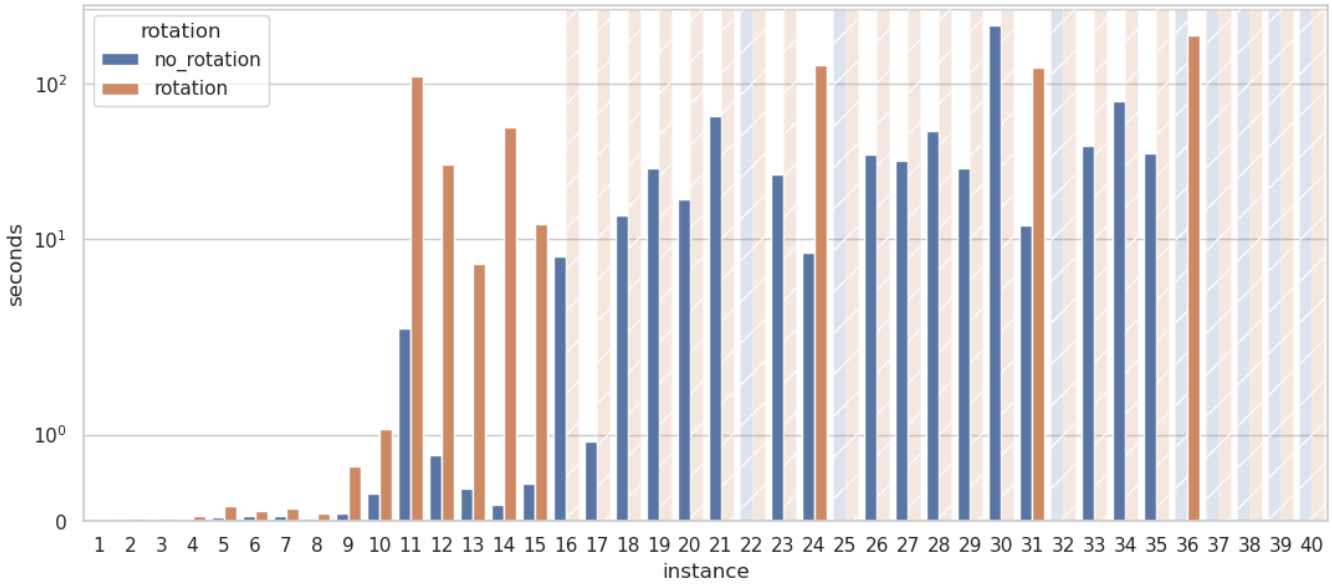

Figure 6: SMT Solving Times

As we can observe, also SMT solves hard instances at least without rotation in a reasonable time. Nevertheless it is almost always outperformed by CP.

5 SAT FORMULATION

SAT formulation could be built quite straightforward starting from SMT. Since chips' positions and dimensions are integer value, we may add a dimension to the arrays, denoting with a single True in a column a certain integer. However this sort of *translation* would lead to a model almost identical to the previous and probably also slower, so not so interesting.

We thought to an alternative way which has shown good performances on simple tasks but some problems appeared in bigger instances. Python code of this proposal is also available on GitHub. These kind of problems do not seem to be deterministic (even if surely they are) and for this reason we only wish to illustrate our idea.

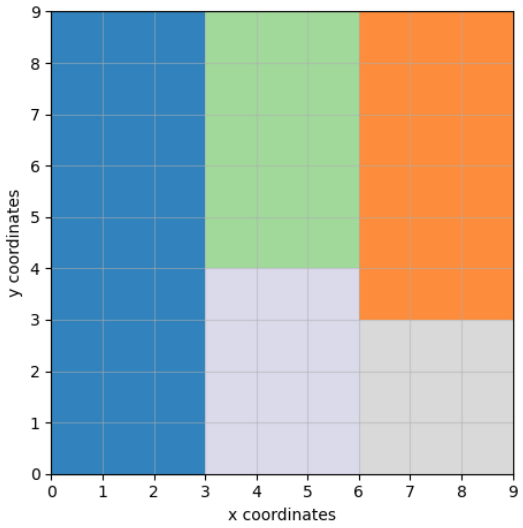

Figure 7: Instance 2 solved with SAT alternative model

First we consider a third dimension z for representing the plate (clearly it is used only during the computation and removed in order to display the solution). The third dimension measures n (one more layer for each chip) and the final result is a parallepiped of $w * h * n$ boolean cells. Every level of cells must contains exactly one chip; k-th level would contain the chip in position k in the width/height arrays. For example a chip 3x2 is represented by 3x2 adjoining True cells while all the other cells of its level are False. The first obvious constraint is not to overlap the chips (rectangles of True over all the different levels) and it can be formulated with a `at_most_one` applied to all the $w * h$ columns. An `at_least_one` is not required since a column made of only False represent an empty square on the plate.

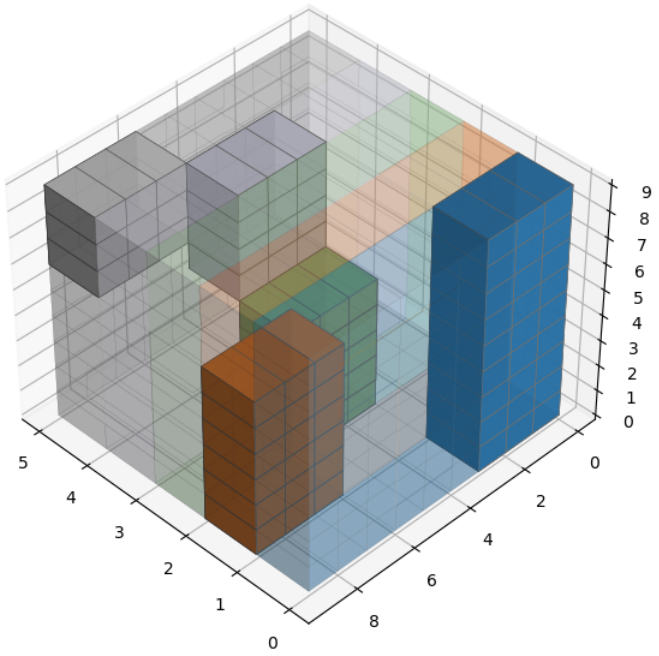

Figure 8: Instance 2 solved with SAT alternative model (3D view)

The second and last constraint is definitely complex to explain, so let's consider a generic k-level (and its relative chip). It must be imposed that, for all the n levels, at least one of the feasible positioning of the k-th chip holds in the k-th level. That means that one among $(w - chips\_w_k + 1) * (h - chips\_h_k + 1)$ legal options for chip k is the right one (according also to the not overlapping constraint), then we can just create these possible truth cells assignment for the level and adding it together as constraint to the solver with an `at_least_one`. Just repeat the procedure in order to add constraints for all the levels.
The SAT Solver will find a consistent truth assignment for all the cells, finally we can construct a solution like fig. 7 projecting the fig. 8 on level 0. Thanks to the different levels from where the projection is applied, the program can know which chip is placed on the planar plate.

6 CONCLUSIONS

Overall we are satisfied of the previously shown results and we tried to give an explanation for the missing typical exponential growth of the computation time for increasing difficulty NP-complete problems. Adding only base constraints (domain and not overlapping constraints) and using an heuristic-less strategy, we can easily verify that instances over the 15th are almost impossible to solve in a reasonable time because the search tends to be exhaustive for each tree level and there can not be peaks of depth. Things changes introducing search heuristics and new constraints that effectively prune the search space; it rises the chance of finding a *good* part of the search tree and enables

relevant depth peaks, in other words a relevant percentage of failure nodes will never be explored.
Starting from these premises we infer that all we have done is making more probable to avoid dead branches but we have no idea of how much more time would require, for example, solving the instance 40. At the same way, overall bad constraints (maybe computationally expensive or not very good at pruning) can modify the searching order of just a single variable, guessing a critical value which leads to an early success. At the end of the day, a good problem model should be evaluated taking in account many instances of high difficulty and with long cut-off time because several single positive outcomes may result from luck.

We finally believe that it would be very interesting keeping to explore the SAT model proposal and integrate restart strategy in both SMT and CP approaches.