UCLouvain

École polytechnique de Louvain

Modeling and Solving Composite Structure Design with Constraint Programming

Author: **Miguel Antoons**
Supervisor: **Pierre Schaus**
Readers: **Augustin Delecluse, Hélène Verhaeghe, Samih Zein**
Academic year 2024–2025
Master [120] in Computer Science

Abstract

Composite Structures are widely used due to their numerous desirable mechanical, cost and weight properties. They are composed of plies (layers) of carbon fibers. For each ply, one must decide its orientation from the set of possible angles: -45°, 0°, 45°, and 90°. The stack of plies must follow strict constraints on the chosen orientations to achieve mechanical properties of the composite, such as sufficient buckling load. The design problem becomes more complex when determining the stack of plies for a complete surface material, that does not require the same number of plies in every region of the surface. Not only must the orientations be selected in each region, but it is also necessary to decide which plies are discontinued between adjacent regions. Thanks to its declarative nature, Constraint Programming (CP) offers an elegant modeling of the constraints, making it easy for designers to activate or deactivate them as needed. This thesis proposes a CP model to solve the composite structure problem. The performance of this model on synthetic yet realistic instances when solved by different exact solvers, including Mixed Integer Programming (MIP) solvers, demonstrates the superiority of CP over MIP and over a commercial solution implemented by an industrial partner. We then propose a few search strategies, using MaxiCP, that can enhance the solving of this problem and study how to find solutions to unfeasible instances of the problem by relaxing some constraints. It opens up the adoption of CP as an efficient building block of Computer-Aided Design tools for composite structures.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisors, Pr. Pierre Schaus and Mr. Augustin Delecluse, for their guidance throughout this project and their assistance in the writing of this thesis.

I also want to thank Dr. Samih Zein for his support and for sharing his expertise on the subject.

Lastly, I extend my thanks to my parents for their unwavering support, which allowed me to do this project as well as everyone who helped me by proof-reading this thesis and giving me valuable feedback on both its content and form.

In the preparation of this thesis, various AI tools were employed to assist in drafting and refining certain sections, ensuring clarity and coherence throughout the document.

Contents

Chapter 1

Introduction

Composite structures are widely used due to their numerous desirable mechanical, cost and weight properties in various fields, including aerospace, automotive, construction, etc. A composite structure is composed of a stack of plies, each ply consisting of fibers (e.g. carbon fibers, glass fibers, ...) and resin. Traditionally, the fibers in these plies follow one of four conventional directions, that is -45°, 0°, 45° or 90°. The plies take different fiber orientations to improve the overall strength and stiffness of the structure by distributing loads more effectively in multiple directions. Composite structures are divided into several zones, also known as stacking sequences, which can have varying thicknesses of plies. This way, weight can be saved by dropping plies in low stress zones and keeping or adding plies in high stress zones. A key challenge in composite structure design is optimizing the design stacking sequence of fiber layers to achieve the best balance between strength, weight, and manufacturability. The angles must be carefully chosen by the designer as they modify the properties of a composite structure. Additionally, the angles are subject to constraints imposed by the manufacturers of such structures. Those constraints exist to warrant robustness, stability and the ability to manufacture these parts. Designers of compostie structures might find it convenient to know if a conceptual composite structure is actually manufacturable, if possible in reasonable time. However, finding a feasible solution becomes increasingly hard as the number of stacking sequences and plies increase.

An example of a composite structure can be seen in figure 1.1. In this case the structure represents a plane wing. We can see a wing divided into five different zones going from the root to the tip of the wing. On the top left we can see the different thicknesses of the sequences, as well as the blending of the plies going from a stacking sequence to another. For a plane wing, it is important to have strength

at the root since it has to carry the weight of the whole structure. However, the tip of the wing carries only a fraction of the weight. Therefore we drop plies going from root to tip, saving weight where possible while keeping strength where needed. Finally, on the right part of the image we can see a detailed stacking sequence (in this case the root stacking sequence), with the different plies and their angles. Of course, it is important to note that this image is a simplification and that real composite structures are much more complex, with more stacking sequences and plies.

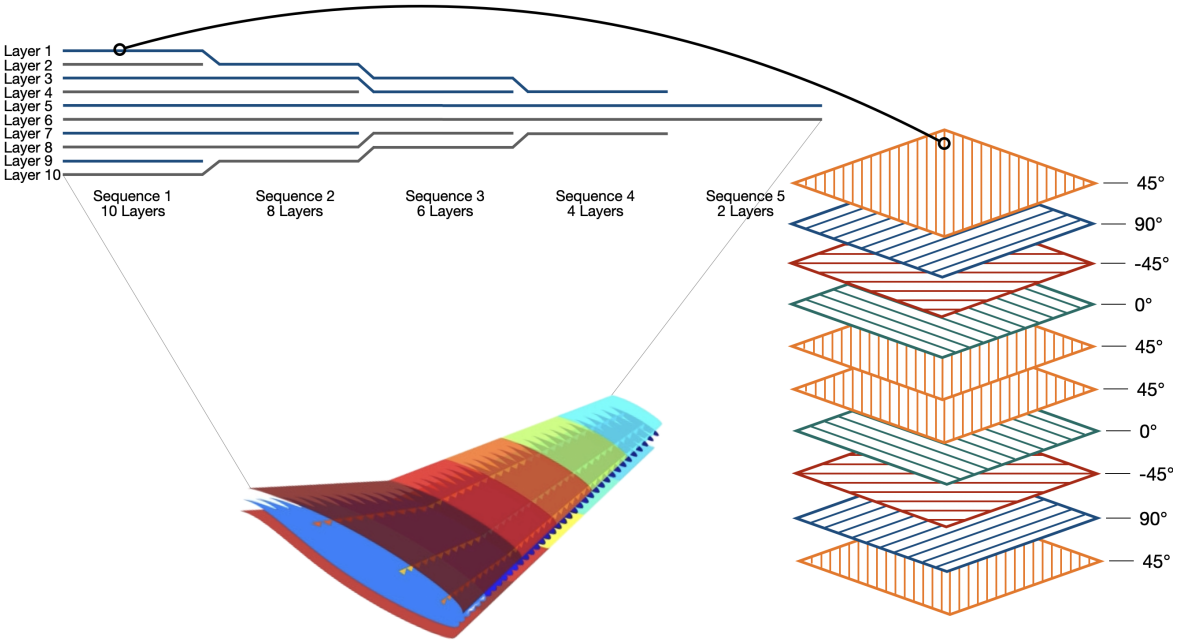

Figure 1.1: Example of the use of composite structures, here with a with a plane wing.

1.1 Problem Definition and Goal

Discovering the optimal stacking of the layers and how they interleave between zones to form the whole structure is a combinatorial optimization problem. A lot of research has been conducted on finding algorithms that manage to find both feasibility and optimilaity of a composite structure design. As can be seen in a review on the optimization of composite structures [26], this type of problem has so far been mainly addressed using genetic algorithms. A notable exception

is the exact algorithm proposed in [42], implemented in the commercial tool of our industrial partner Cenaero, and which we compare with. Other approaches mainly use meta-heuristics for solving those problems but without any optimality guarantees.

The goal of this research project is to improve this process by using exact methods, specifically Constraint Programming (CP), instead of traditional meta-heuristic approaches. More precisely, this thesis aims to create and test a model in Constraint Programming to evaluate its potential in designing composite structures. The results of this thesis were also written in the following research paper: Antoons Miguel, Delecluse Augustin, Schaus Pierre, and Zein Samih. "Modeling and Solving a Composite Structure Design Problem with Constraint Programming". In: *CP2025* (2025).

After a brief definition of constraint programming in section 1.2, this research document is organized as follows. Chapter 2 first presents the problem definition for deriving stacking sequences of a composite structure. A Constraint Programming model is then introduced in Chapter 3. This is followed by Chapter 4 which presents different search strategies that can be used for this problem. Then, in Chapter 5 existing solutions and differences with our implementation will be discussed. In Chapter 6 our model will be tested and the different search strategies will be evaluated. This is followed by Chapter 7 which studies how we can relax constraints to find solutions to unfeasible instances of the problem. Chapter 8 then discusses several potential improvements that could be added to our model in future works. Finally, 9 summarizes and concludes this thesis.

1.2 Constraint Programming

Constraint Programming is a programming paradigm that allows to describe and solve Constraint Satisfaction Problems (CSP). A Constraint Satisfaction Problem is a mathematical problem defined as a set of objects whose state must satisfy a number of constraints or limitations. It consists of three parts. A set of variables $X = \{x_1, ..., x_n\}$, a finite domain for every variable $dom(x_i) \subseteq D$ and a collection of constraints C. Each constraint C_i is applied over some subset of the variables in X and specifies the allowed combination of variables that this subset can take. A solution to a CSP is a combination of assignments to every variable such that all constraints are satisfied.

Example 1.2.1. Suppose a CSP consisting of two variables $X = \{x_1, x_2\}$. The

domains of variables x_1 and x_2 consists of the whole set D: $dom(x_1) = dom(x_2) = \{0, 1, 2, 3, 4\}$ and there is a single constraint c_1 that enforces $x_1 < x_2$. In other words, the solution space of this CSP consists of all the assignments where the value of x_1 is strictly lower than the value of x_2. The valid combinations of values that satisfy the c_1 constraint are listed in the table below.

x_1	0	0	0	0	1	1	1	2	2	3
x_2	1	2	3	4	2	3	4	3	4	4

Constraint Programming offers a way to define and solve those problems in two steps. The first step consists of creating a model representing the CSP. This means: declaring the variables, their domains and the constraints that need to be satisfied. Moreover, the constraints present in CP frameworks, also called CP solvers, often implement filtering algorithms to reduce the domain of the variables. For instance, in our example 1.2.1 we notice from the start that the variable x_1 will never be able to take the value 4 since the maximum value in x_2 is 4. The same goes for variable x_2 which will never be able to take the value 0 since the minimum value in x_1 is 0. Filtering algorithms will take care of those values and prune them to reduce the search space.

This takes us to the second step used in CP solvers: the search for a solution. In this step, the solver constructs a search tree by by changing, at each step, the domain of a variable. This change can be an assignment of a variable to a value, the removal of a value from a variable domain, etc. If the assignment violates a constraint, the search algorithm backtracks and tries to assign a different value. We can easily see that the size of the search tree grows with the number of variables as well as the size of the domains from these variables. It is therefore essential that good filtering algorithms are implemented to reduce the domains as much as possible. An example of a search tree of the problem described at example 1.2.1 can be seen in figure 1.2. Note that the search algorithm is able to find all solutions that respect the c_1 constraint. Of course our filtering algorithm is strong enough to filter all values leading to unsatisfiable solutions, so no backtracks are present here. However, on more complex problems, such as verifying feasibility for composite structures, filtering algorithms are not able to filter all infeasible values, leading to many infeasible solutions, more backtracks and thus, a larger search tree.

In this thesis two CP technologies will be used in order to complete our goal. The first one is MiniZinc [24]. MiniZinc is a high-level constraint modelling language. It allows users to define their problems in a declarative way, focusing on the "what" rather than the "how". This means that you specify the constraints and objectives of the problem without worrying about the underlying algorithms used to solve it.

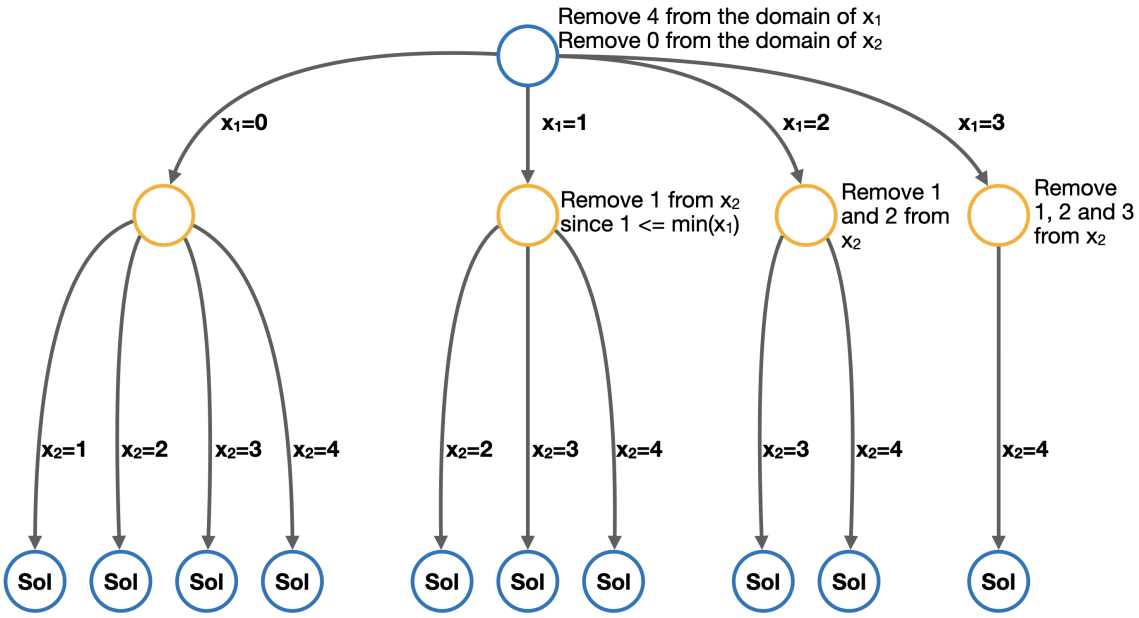

Figure 1.2: A search tree from the CSP described at example 1.2.1.

MiniZinc models can be solved using various solvers, such as Gecode, Chuffed, and Google OR-Tools. This allows us to experiment with different solvers in an abstract way, that is, without having to know the specifics of each solver. We will use MiniZinc to test our model against different solvers as well as to compare it with the solution of our industrial partner, without worrying about the implementation details.

The second tool we will use is MaxiCP [37]. MaxiCP is an in-development solver based on MiniCP [22] and has recently been released open-source. It is developed at UCLouvain and allows full control of the implementation of the different algorithms. This means that the filtering algorithms can be modified as wished and custom search strategies can be implemented. In this thesis, MaxiCP will serve to compare different existing and custom search strategies, as this is not possible in MiniZinc.

Chapter 2

Rules

As was mentioned before, the creation of composite structures must respect a set of rules. The problem is to decide the orientation of each ply in each stacking sequence while satisfying rules specific to each stacking sequence and at the same time connecting those rules between adjacent sequences. Those rules can be divided into two distinct categories, namely the design rules and the blending rules. The design rules will be denoted by D1 to D5 while the blending rules will be denoted by B1 to B4. Depending on the structure requirements, some rules may be ignored. All blending rules B1-B4 are always present, to ensure a global coherence between adjacent sequences. The design rules D1-D5 may be absent in some settings according to the requirements of the customer. This further motivates the use of declarative paradigms such as CP to tackle this problem. In this chapter, those two sets of rules will be described as well as some notations that will be used in the remainder of this document.

2.1 Design Rules

The design rules constraint how the plies can be laid on top of each other in a particular stacking sequence S_i without worrying about the stacking sequences next to it. In this section, those rules will be presented in detail.

2.1.1 D1: Imposed Ply Cardinalities

Three cardinalities are given for every stacking sequence S_i: the number of plies of 0° that must be present in the sequence, the number of plies of -45°/45° that must be present in a sequence and the number of plies of 90° that must be present in a sequence. Those cardinalities will be named c_i^0, $c_i^{+/-45}$ and c_i^{90} throughout

this document, while the number of plies of each distinct angle that is present in a sequence will be denoted by n_i^{-45}, n_i^0, n_i^{45} and n_i^{90}. Those cardinalities have to be respected in order to generate feasible stacking sequences. The thickness of each sequence, denoted by n_i must be equal to the sum of those cardinalities. This means that $n_i = c_i^0 + c_i^{+/-45} + c_i^{90} = n_i^{-45} + n_i^0 + n_i^{45} + n_i^{90}$ and $c_i^{+/-45} = n_i^{-45} + n_i^{45}$.

Note that there is only one cardinality given for the -45° and the 45° plies ($c_i^{+/-45}$). This is because the sequence must be balanced, meaning that the number of plies taking the -45° angle must be equal to the number of plies taking the 45° angle. Therefore, the $c_i^{+/-45}$ regroups the number of -45° plies and the number of 45° plies. Of course, when the cardinality is uneven, it is allowed that the amount of -45° and 45° plies differ by one.

In addition to the given cardinalities, we must ensure that there is a minimum of 8% of plies of every angle in each stacking sequence at all times. While this is not a problem when the D1 constraint is activated (since we can assume that the given cardinalities respect this), this constraint must be enforced in case the D1 constraint is not activated. Otherwise, suppose that this constraint is deactivated, we could simply assign the same angle to every ply and have a feasible solution. Therefore, a constraint assuring that the minimum of 8% of plies of every angle in each stacking sequence is are respected will be active when the D1 constraint is deactivated.

2.1.2 D2: No Angle Difference of 90°

The angle difference between two consecutive plies cannot be equal to 90°. This means that a -45° ply cannot be followed by a 45° ply, a 0° ply cannot be followed by a 90° ply and the inverse of those two cases is also prohibited.

2.1.3 D3: Maximum 3 Consecutive Plies of Identical Angle

At every point in every stacking sequence, there must not be a succession of four or more plies having the same angle. This number of four differs depending on the source, the manufacturer and the customer. We chose a maximum number of three in accordance with our industrial partner to test our model.

2.1.4 D4: Vertical Symmetry of the Stacking Sequences

Every stacking sequence must be vertically symmetric. This means that the top ply must be identical to the bottom ply, the second ply must be identical to the

penultimate ply and so on. This goes on until we reach the middle two plies for even thickness sequences or the middle three plies of odd thickness sequences. This is to account for cases for which it is not possible to have an even number of plies for all angles. In those cases we allow a dissymmetry of $(-45|45)$ for the middle two plies in the case of an even thickness sequence and we allow a dyssymmetry of $(-45, 0, 45)$ and $(-45, 90, 45)$ for the middle three plies in the case of an odd thickness sequence. In figure 2.1 we can see the allowed middle dyssymmetries. Note that their inverses are also allowed.

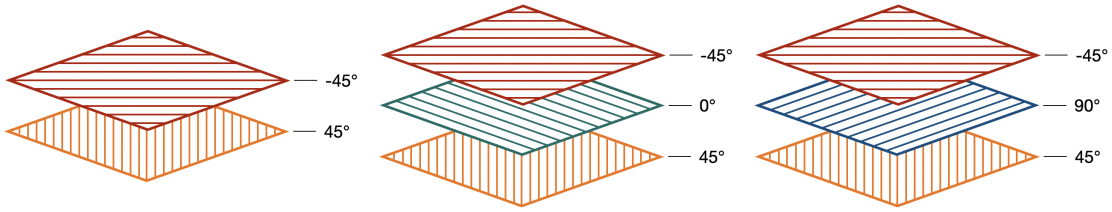

Figure 2.1: Allowed dissymmetries in the middle of stacking sequences.

2.1.5 D5: Limited Angles for Top and Bottom Plies

The top and bottom plies of every stacking sequence are only allowed to take their angles from the set $\{-45°, 45°\}$.

2.2 Blending Rules

A second set of constraints consists of the blending rules, which restrict how neighboring stacking sequences relate to each other. Composite structures are translated into a graph so that they can be used as input for our model. In this graph, every stacking sequence represents a node and adjacent nodes are connected with an edge. For every pair of neighboring stacking sequences S_i and S_j, an implicit directed edge points from the sequence with the larger number of plies toward the one with fewer plies ($n_i > n_j$). We call i a parent of j. This directed graph represents the neighboring relationships. This graph must be acyclic for the structure to be feasible. The set of edges in this graph is denoted E. Blending rules are always enforced.

2.2.1 B1: Blending In Between Stacking Sequences

The stacking sequence S_j (the thinner one) must be a subsequence of S_i (the thicker one). In the event S_j has multiple parents, its plies must be contained in all of

them. For instance, in figure 2.2 we can see on the right an instance that respects this rule and on the left an instance that violates this rule. Indeed, while in the left example the middle sequence takes it plies from its parents, it is not a subsequence of both its parents. For it to be correct, the plies should continue from one parent to the other, as is the case on the right example.

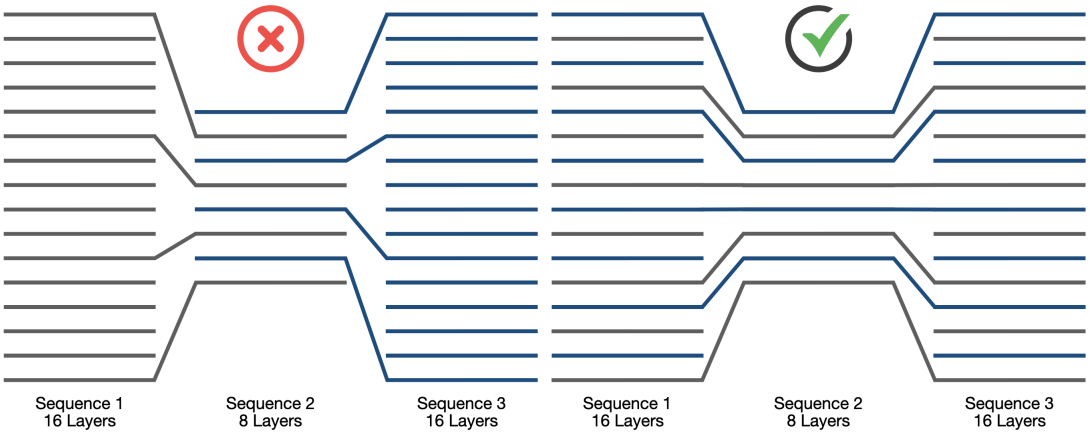

Figure 2.2: An example of incorrect and correct blending with multiple parents.

2.2.2 B2: No Ply Crossing Allowed

A ply can either be shared between two neighboring stacking sequences or be discontinued, but two continuous plies cannot cross each other. This would otherwise result in structures that are hard to manufacture and have critical weak points. An incorrect and correct implementation of this rule is shown in figure 2.3.

2.2.3 B3: Surface Plies Must Be Continuous

The surface plies must be continuous across the entire structure. The example shown in figure 2.4 violates this rule since the surface plies of the first three sequences is not identical to the surface plies of the fourth stacking sequence.

2.2.4 B4: Maximum Ply Drop-off In Between Stacking Sequences

A maximum of three consecutive layers can be dropped at once from a parent sequence S_i to a child sequence S_j. This rule is shown in figure 2.5.

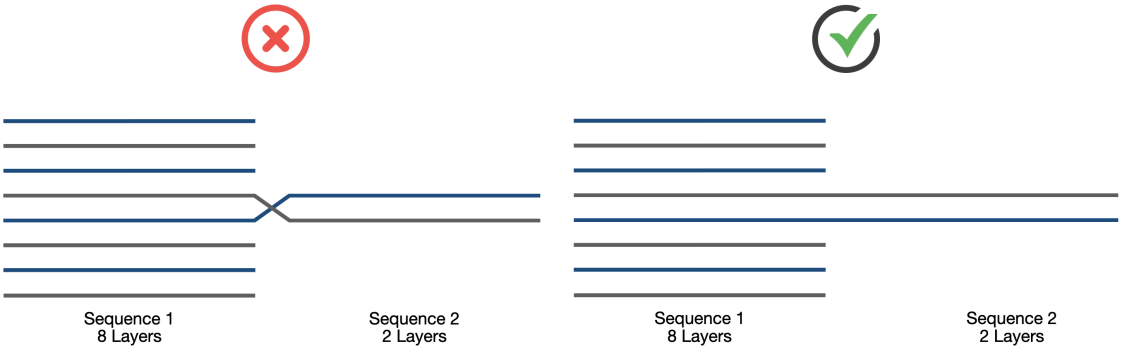

Figure 2.3: Example that violates and an example that implements the rule disallowing plies from crossing each other.

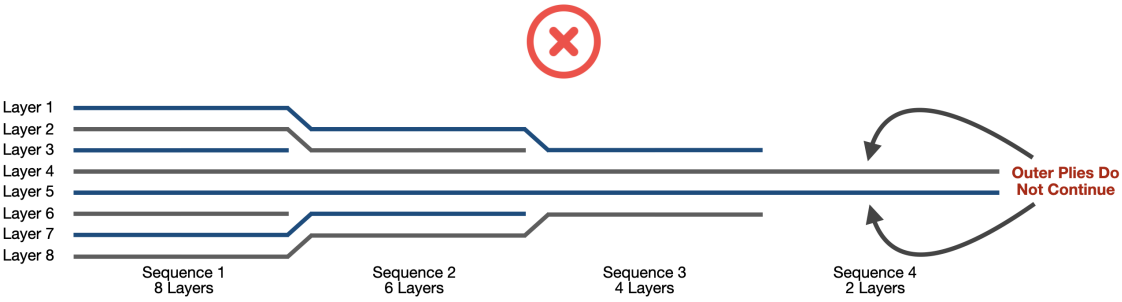

Figure 2.4: An example of a structure that violates the rule stating that outer plies should be continuous.

2.3 A Visual Example

In figure 2.6, a composite plate containing four different zones is transformed into a graph that can be used as input for our model. This graph contains four stacking sequences (equal to the number of zones) and the cardinalities are given for each one of them. Given the input graph from figure 2.6, a solution satisfying most of the constraints is shown in figure 2.7. Note that sequence S_1 is shown twice in this image. This is to better show the blending of the plies from S_1 to S_3. In this solution, the constraints satisfied are B1, B2, B3, B4 and D1, which are always activated as well as the optional constraints D3, D4 and D5. There are some places where an angle difference of 90° is present, which is prohibited by constraint D2. This is for instance the case for $S_1[2] = 90°$ and $S_1[3] = 0°$.

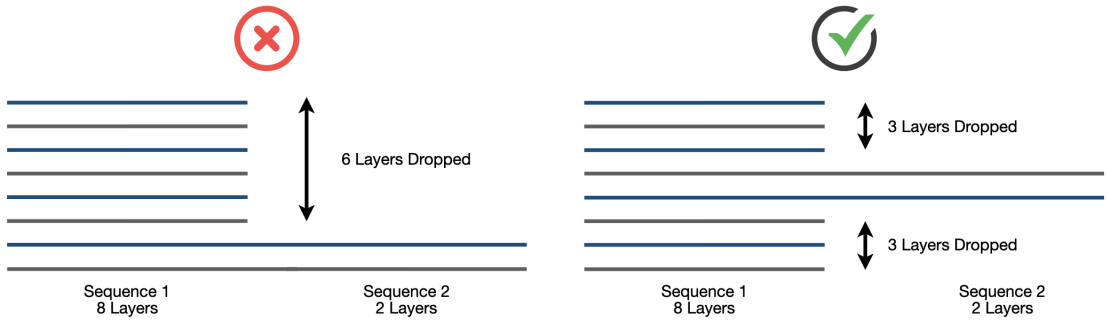

Figure 2.5: Image showing an example that violates and an example that implements the rule disallowing 4 or more consecutive plies from being dropped.

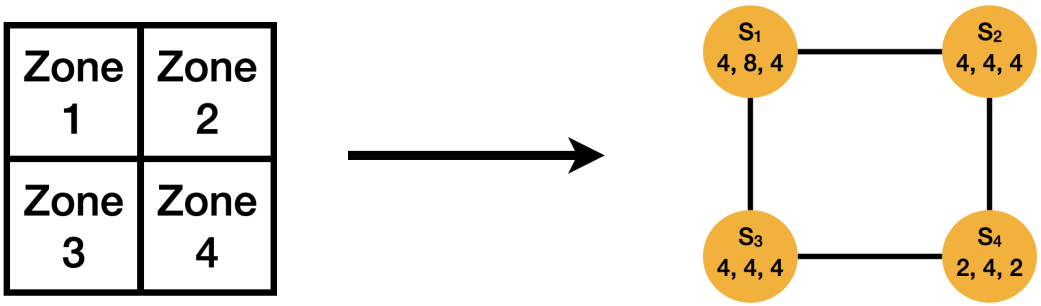

Figure 2.6: An example of how we represent composite structures for our CP model.

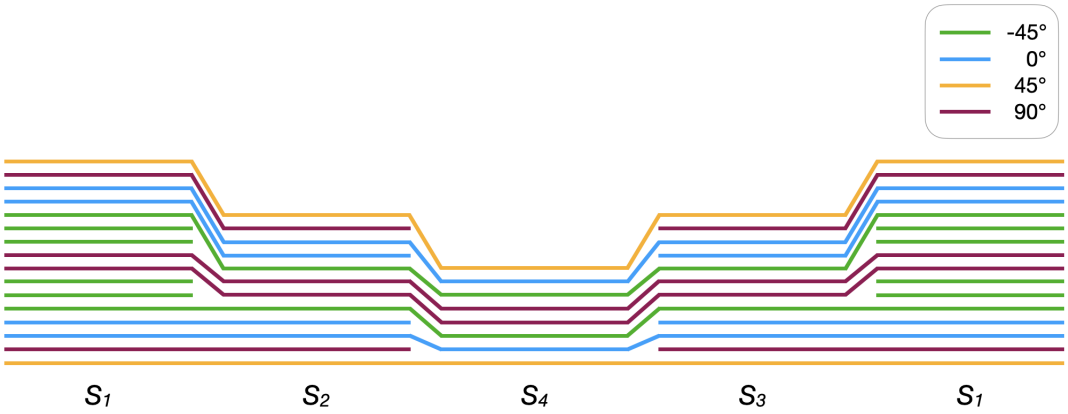

Figure 2.7: An example solution of our model with a limited set of constraints activated.

Chapter 3

Model

In this chapter, two topics will be discussed. First, in section 3.1 our Constraint Programming model will be presented and the choice of our constraints will be justified. Then, in section 3.2 the MiniZinc code of our model will be shown and explained, allowing to get a complete overview of it.

3.1 Constraints

We will now go over the different rules and translate them into constraints that can be used in a Constraint Programming framework. For each constraint of our model, the mathematical formulation will be provided. At the end of this chapter, the MiniZinc code of our entire model is shown in figure 3.3 as well as an input file in figure 3.2.

3.1.1 Variables

The main decision variables of our problem are the orientations that each layer k can take in every stacking sequence S_i:

$$S_i[k] \in \{Id_{-45},\ Id_0,\ Id_{45},\ Id_{90}\} \qquad \forall\ 1 \leq k \leq n_i \qquad \forall i \in \mathcal{I} \tag{3.1}$$

Where $Id_a, a \in \{0°, \pm 45°, 90°\}$ denotes the plies orientation, $\mathcal{I} = \{1 \dots |S|\}$ is the set of indices for all stacking sequences S. A second set of variables $Y_{i,j}[k] \in \{1, \dots, n_i\}$, for all $(i,j) \in E$ is used to ensure the continuity of layers between adjacent regions. More precisely $Y_{i,j}[k]$ indicates that the ply k of the stacking sequence j is the continuation of the ply $Y_{i,j}[k]$ in the neighboring stacking sequence i. For example, referring to Figure 2.6 and 2.7, $Y_{1,2}[6] = 8$ since the sixth

ply from S_2 is the same as the eighth ply from S_1.

The structural constraints D1-D5 on each stacking sequence are expressed on $S_i[k]$ variables while the blending constraints B1-B4 are expressed on $y_{i,j}$ variables. The formal definition of the constraints is given next.

3.1.2 D1: Cardinality Constraint

The D1 rule, enforcing a number of plies for the 0°, +/-45° and 90° angles, given by the cardinalities at the beginning of our model in c_0, $c_{-45/45}$ and c_{90}, can be enforced using a global cardinality constraint [36]. A global cardinality constraint specifies that each value in a set must appear a given number of times within a set of variables. Different types of this constraint exist, in our case we will use two of them: one to enforce a lower bound on the ply cardinalities and one to enforce an upper bound on the cardinalities. Those two constraints will be named gc_min and gc_max for the lower and upper bounds respectively.
The formal definition of this constraint on every stacking sequence S_i is the following.

$$min45 := \lfloor c_{-45/45}/2 \rfloor \tag{3.2}$$
$$max45 := \lceil c_{-45/45}/2 \rceil \tag{3.3}$$
$$gc\_min(S_i, [Id_{-45}, Id_0, Id_{45}, Id_{90}], [min45, c_0, min45, c_{90}]) \tag{3.4}$$
$$gc\_max(S_i, [Id_{-45}, Id_0, Id_{45}, Id_{90}], [max45, c_0, max45, c_{90}]) \tag{3.5}$$

We first define two variables in 3.2 and 3.3. They are the lower and upper bounds respectively for the -45° and 45° cardinalities. The reason we *floor* and *ceil* those values is to account for the case where $c_{-45/45}$ is odd. Then, in 3.4 we first initialize the constraint enforcing minimum cardinalities, followed by the constraint enforcing maximum cardinalities in 3.5. These constraints take three arguments:

1. The first one is the sequence itself, composed of all the ply variables that are contained in it. Each of the variables is able to take Id_{-45}, Id_0, Id_{45} or Id_{90} as its value.

2. The second argument is an array with the values whose cardinalities must be constrained in the sequence that was given as the first argument. For both of the constraints listed above, those values are Id_{-45}, Id_0, Id_{45} and Id_{90} which represent the angles -45°, 0°, 45° and 90° respectively.

3. The third argument is an array containing the cardinalities of every value given in the second argument. So for instance, the number written at the first

place of this argument is the minimum (respectively maximum) amount of the value located at the first place of the array given in the second argument for the *gc_min* (respectively *gc_max*) constraint.

The reason we use two constraints instead of a single one is because the cardinalities for the -45° and 45° plies are not given exactly at the start of the program. Therefore we cannot specify precise cardinalities for every angle. Instead of this, we allow a range for the -45° and 45° angles, allowing them to differ by a single ply, and specify an exact cardinality for the 0° and 90° angles.

3.1.3 D2 & D3: No 90° Gap and Maximum 3 Consecutive Identical Plies

Many constraints can be expressed in different ways. Expressing a constraint in a different way can, according to the use case and the solver, improve or worsen the performance of the Constraint Programming model. This is the case for the D2 and the D3 constraints. The simplest expression of the D2 constraint is as written as:

$$|S_i[k] - S_i[k+1]| \neq 90 \qquad \forall\ 1 \leq i \leq n_i - 1 \qquad \forall \mathcal{I}$$

We just take the absolute difference of two consecutive angles and prohibit the result from being equal to 90°. For the D3 rule, For the D3 rule, the simplest expression to prohibit four or more consecutive plies with identical angles, is written as:

$$S_i[k] \neq S_i[k+1] \vee S_i[k] \neq S_i[k+2] \vee S_i[k] \neq S_i[k+3] \qquad \forall\ 1 \leq i \leq n_i - 3 \qquad \forall \mathcal{I}$$

Which states that among four consecutive plies, at least one must be different from the three others. We apply this on every set of four consecutive plies in every stacking sequence.

The D3 rule in particular is complicated since it is a combination of four constraints (three "not equal" constraints and one "or" constraint). Moreover, this constraint can easily be replaced by a table constraint [4, 29, 8, 17]. A table constraint defines a set of tuples, where each tuple represents a valid combination of values for a set of variables. The table constraint ensures that the variables can only take on values that correspond to one of the tuples in the table. In our case, the set of tuples contains every sequence of four angle values (Id_{-45}, Id_0, Id_{45} or Id_{90})

where at least a single value is different from the others. For instance, the sequence $[Id_{-45}, Id_0, Id_0, Id_{-45}]$ is contained in this set, while the sequence Id_0, Id_0, Id_0, Id_0 is not contained in this set. We call this set allowed_tuples. Note that this set will not be shown here as it is too large to display. If we apply this constraint on every set of four consecutive plies in every stacking sequence, as was done before, we get the following.

$$(S_i[k], S_i[k+1], S_i[k+2], S_i[k+3]) \in \text{allowed\_tuples} \quad \forall 1 \leq k \leq n_i - 3, \forall i \in \mathcal{I}$$

If we think a little further, we can see that this specific constraint can also serve to enforce the D2 rule, prohibiting 90° angles in between consecutive plies. Indeed, if we remove tuples where any two consecutive plies contain a 90° angle in addition to removing tuples where all the plies have the same angle, we get a single table constraint that enforces both the D2 and the D3 rule. Doing this would also remove tuples such as $[Id_{-45}, Id_{45}, Id_0, Id_{-45}]$ or $[Id_{-45}, Id_0, Id_0, Id_{90}]$ for instance. The impact of these different formulations will be studied later on in this document.

3.1.4 D4: Vertical Symmetry of the Stacking Sequences

The D4 rule, enforcing stacking sequences to be vertically symmetric can be enforced using an equality constraint. To ease the constraint notations here, we first define some set of indices and constants. $\mathcal{I}_{even} = \{i \in \mathcal{I} \mid n_i \text{ is even}\}$ and $\mathcal{I}_{odd} = \{i \in \mathcal{I} \mid n_i \text{ is odd}\}$ are the sets of indices for sequences of even and odd length, respectively. Moreover, $m_i := \lfloor n_i/2 \rfloor$ denotes the midpoint of a sequence $S_i \in S$. With all this defined, the vertical symmetry can be defined as follows.

$$S_i[k] = S_i[n_i - k + 1] \qquad \forall 1 \leq k < m_i, \forall i \in \mathcal{I}$$

The reason we dont enforce the equality constraint on the middle two or three plies, according to if the structure is of even or odd thickness respectively, is to allow for the middle dissymmetry explained previously. However, we cannot just put any combination of plies in the middle. To limit the combination of plies allowed in the middle to those that are authorized by the rules, we add another table constraint as follows.

$$(S_i[m_i], S_i[m_i+1]) \in \text{allowed\_even} \qquad \forall i \in \mathcal{I}_{even} \tag{3.6}$$
$$(S_i[m_i], S_i[m_i+1], S_i[m_i+2]) \in \text{allowed\_odd} \qquad \forall i \in \mathcal{I}_{odd} \tag{3.7}$$

Constraint 3.6 is for stacking sequences with an even number of plies and constraint 3.7 is for stacking sequences with an odd number of plies. The sets of allowed combinations of middle plies, denoted by allowed_even and allowed_odd are shown in figure 3.1.

$$\text{allowed\_even} := \{[Id_{-45}, \; Id_{-45} \;], \; [Id_0, \; Id_0 \;], \; [Id_{45}, \; Id_{45} \;], \; [Id_{90}, \; Id_{90} \;], \; [Id_{-45}, \; Id_{45} \;], \; [Id_{45}, \; Id_{-45} \;]\}$$

$$\begin{aligned}\text{allowed\_odd} := \{&[Id_{-45}, \; Id_0, \; Id_{-45} \;], \; [Id_{-45}, \; Id_{90}, \; Id_{-45} \;], \; [Id_0, \; Id_0, \; Id_0 \;], \; [Id_0, \; Id_{90}, \; Id_0 \;], \\ &[Id_{45}, \; Id_0, \; Id_{45} \;], \; [Id_{45}, \; Id_{90}, \; Id_{45} \;], \; [Id_{90}, \; Id_0, \; Id_{90} \;], \; [Id_{90}, \; Id_{90}, \; Id_{90} \;], \\ &[Id_{-45}, \; Id_0, \; Id_{45} \;], \; [Id_{45}, \; Id_0, \; Id_{-45} \;], \; [Id_{-45}, \; Id_{90}, \; Id_{45} \;], \; [Id_{45}, \; Id_{90}, \; Id_{-45} \;]\}\end{aligned}$$

Figure 3.1: The allowed combination of plies in the middle of a stacking sequence.

3.1.5 D5: Limited Angles for Top and Bottom Plies

To limit the angles of the top and the bottom plies to only $\{-45°, 45°\}$ we can simply redefine the domain of the top and bottom plies in all the stacking sequences. This can be written as follows.

$$S_i[1] \in \{Id_{-45}, Id_{45}\} \qquad \forall i \in \mathcal{I}$$
$$S_i[n_i] \in \{Id_{-45}, Id_{45}\} \qquad \forall i \in \mathcal{I}$$

While one might think that redefining the domain of only the top ply might be enough, given the vertical symmetry constraint, we have to remember that constraints can be deactivated according to the wishes of the customer.

3.1.6 B1: Blending In Between Stacking Sequences

For the blending constraint, we can use element constraints [12]. The element constraint establishes a relationship between an index variable, an array of values,

and a target variable. It ensures that the target variable takes on the value from the array at the position specified by the index variable. For the B1 rule, we want that every child sequence takes all of its plies in its adjacent parent sequences. Therefore, the target variable is a ply variable of a child sequence and the array of values is the array of ply variables that represents the parent sequence. Moreover, if it has multiple parents, the plies must go from one parent, through the child sequence, to the other parent. This constraint can be described in a more formal way as follows.

$$S_j[k] = S_i[Y_{i,j}[k]] \qquad \forall 1 \leq k \leq n_j, \quad \forall (i,j) \in E$$

We can see that an element constraint is applied to every ply of a child sequence, for every pair of adjacent stacking sequences. Indeed, E represents the edges of our input graph and every edge connects a parent (sequence i) to a child (sequence j). There exists one index variable, $Y_{i,j}[k]$, for every ply of a subsequence, for every edge. This variable will eventually contain the index of the ply that the sequence S_j takes from S_i.

3.1.7 B2: No Ply Crossing Allowed

A simple way to avoid plies from crossing each other in between sequences is to apply a constraint on the index variables Y, used in the B1 constraint. Indeed, if we simply force each index variable of a ply ($Y[k]$) to be strictly lower than the index variable of the next ply ($Y[k+1]$) in the same stacking sequence, plies will not be allowed to cross each other. This is described in the following constraint.

$$Y_{i,j}[k] < Y_{i,j}[k+1] \qquad \forall 1 \leq k < n_j, \quad \forall (i,j) \in E$$

This constraint is applied on every ply of a subsequence for every pair of adjacent stacking sequences.

3.1.8 B3: Surface Plies Must Be Continuous

To enforce the B3 rule, requiring that the surface plies be continuous, we make sure that for every pair of adjacent stacking sequences, the top and bottom plies of the child sequence are the same as the top and bottom plies of the parent sequence respectively. This is done by fixing the index variables of the top and bottom plies as follows.

$$Y_{i,j}[1] = 1 \qquad \forall (i,j) \in E$$
$$Y_{i,j}[n_j] = n_i \qquad \forall (i,j) \in E$$

We do this for every edge in the input graph.

3.1.9 B4: Maximum Ply Drop-off In Between Stacking Sequences

To ensure that there are no more than three consecutive plies that are dropped in between two stacking sequences, we once again constraint the index variables of the subsequences. Indeed, if we make sure that the gap between two consecutive plies that are blended from the parent sequence to a chile sequence does not exceed three plies, this rule is satisfied. This is exactly the same as making sure that the difference between two consecutive index variables is less or equal to four. This is described in the following constraint.

$$Y_{i,j}[k+1] - Y_{i,j}[k] \leq 4 \qquad \forall 1 \leq k < n_j, \quad \forall (i,j) \in E$$

This is then done for every pair of consecutive ply index variables and for every edge in the input graph.

3.2 MiniZinc Model

In this section, we will go over the entire MiniZinc model as used for the CP-SAT and Gecode solvers, including an input file. Note that all the models and written code that was used for the creation of this thesis can be found here, with a description of the file structure here. The input file presented in figure 3.2 represents the graph that was used in the visual example in section 2.3. Several variables are assigned in this input file:

- `T` is the thickness of the thickest stacking sequence in the whole structure.
- `n` is the amount of stacking sequences in the structure, and thus also the amount of nodes in the graph.
- `m` is the amount of edges in the input graph.
- `edges` contains, as its name suggests, the edges from the input graph. An edges always starts at the supersequence and goes to a subsequence.

- `counts` contains the cardinalities for every node. Note that here the cardinalities are given for every angle. In the program, the cardinalities for the -45° and 45° angles will be additionned together.

```
1 T = 16;
2 n = 4;
3 m = 4;
4 edges = [(1, 2), (1, 3), (2, 4), (3, 4)];
5 counts = [|4, 4, 4, 4|2, 4, 2, 4|2, 4, 2, 4|2, 2, 2, 2|];
```

Figure 3.2: An input of our model corresponding to the graph from our example in section 2.3

This is then used as input for our model, shown in figures 3.3, 3.4 and 3.5. At the start all the variables are defined and some of them are already assigned. Note that here, the top line defines $Id_{-45} = 1$, $Id_0 = 2$, $Id_{45} = 3$ and $Id_{90} = 4$.

Then, the MiniZinc model starts with the `P1`, `P2` and `P3` constraints. Those constraints only serve to mitigate some limitations of the MiniZinc framework and are not important to understand the model. The design constraints then start at line 50. They are almost exactly the same as specified above, except for a few details:

- The D1 constraint uses a single cardinality constraint that allows to give lower and upper bounds simultaneously instead of using two distinct cardinality constraints.
- The table constraint enforcing both the D2 and D3 rules is relaxed in the middle to allow for the middle dissymmetries such as $(-45|45)$. This is done by applying a slightly different table constraint in the middle only, allowing 90° gaps.
- For the D5 rule, instead of redefining the domain, we simply prohibit the variables from taking the values that are not allowed.

Note also that the table constraint used for the D2 and the D3 constraint disappears for the Chuffed solver and instead two different constraints are used, following what was said in section 3.1.3

Finally, at line 91 the blending constraints are defined. They correspond exactly to the blending constraints defined above.

```
% define set of possible directions with 0 = /, 1 = -45, 2 = 0, 3 = 45, 4 = 90
set of int: Directions = 0..4;
% define the cover
array[1..4] of int: cover = [1, 2, 3, 4];
% define all the allowed tuples
array[_, 1..4] of int: allowed = ...;
array[_, 1..4] of int: allowed_middle = ...;
% define allowed middle in the case of even thickness
array[1..6, 1..2] of int: allowed_pair = ...;
% define allowed middle in the case of odd thickness
array[1..12, 1..3] of int: allowed_odd = ...;
% minimum ply percentage for every stacking sequence
float: min_ply_percentage = 0.08;

% INPUT
% define the size T of the thickest stacking sequence
int: T;
% define the number of stacking sequences
int: n;
% define number of edges
int: m;
% define a 2D array with the stacking sequences
array[1..n, 1..T] of var Directions: seq;
% define an array with pointers to the parent sequence (used in element constraint)
array[1..m, 1..T] of var 1..T: indexes;
% define the edges between the stacking sequences, the first element is the parent and
% the second is the child
array[_] of tuple(int, int): edges;
% define the number of plies in each direction (used for both the no 90 gap constraint
% and the max 4 consecutive plies in the same direction constraint)
array[1..n, 1..4] of int: counts;
% define a list with the thickness of each sequence, thickness[1] is the thickness of S1
array[1..n] of int: thickness = array1d(1..n , [counts[i, 1] + counts[i, 2]
    + counts[i, 3] + counts[i, 4] | i in 1..n]);

```

Figure 3.3: The complete code of our model in MiniZinc (1/3).

```
36
37 % CONSTRAINTS
38 % MANDATORY CONSTRAINTS
39 % P1 -- for each stacking sequence, make sure that the ply can't take the 0 value when
40 % within the thickness
41 constraint forall(i in 1..n, j in 1..thickness[i])
42     (seq[i, j] != 0);
43 % P2 -- for each stacking sequence, set all the variables after the thickness to 0
44 constraint forall(i in 1..n, j in thickness[i]+1..T)
45     (seq[i, j] = 0);
46 % P3 -- indexes must be lower than the thickness of the parent sequence
47 constraint forall(i in 1..m, j in 1..thickness[edges[i].2])
48     (indexes[i, j] <= thickness[edges[i].1]);
49
50 % DESIGN CONSTRAINTS
51 % D1 -- a given number of plies in each direction is given
52 constraint forall(i in 1..n)
53     (global_cardinality_low_up(
54         seq[i, ..],
55         cover,
56         [floor((counts[i, 1] + counts[i, 3]) / 2), counts[i, 2],
57             floor((counts[i, 1] + counts[i, 3]) / 2), counts[i, 4]],
58         [ceil((counts[i, 1] + counts[i, 3]) / 2), counts[i, 2],
59             ceil((counts[i, 1] + counts[i, 3]) / 2), counts[i, 4]]
60     ));
61 % D2 & D3 -- table constraint containing the no 90 gap contraint and the max 3
62 % consecutive plies in the same direction constraint
63 constraint forall (i in 1..n, j in 1..thickness[i]-3) (
64     if j < floor(thickness[i]/2) - 2 \/ j > ceil(thickness[i]/2) then
65         (table([seq[i, j], seq[i, j+1], seq[i, j+2], seq[i, j+3]], allowed))
66     else
67         (table([seq[i, j], seq[i, j+1], seq[i, j+2], seq[i, j+3]], allowed_middle))
68     endif
69 );
70 % D4.1 -- symmetric sequences: for every node A with thickness t: ai = at-i (except for
71 % the middle part, see next constraint)
```

Figure 3.4: The complete code of our model in MiniZinc (2/3).

```
constraint forall(i in 1..n, j in 1..floor(thickness[i]/2)-1)
    (seq[i, j] = seq[i, thickness[i]-j+1]);
% D4.2 -- allow dysymmetry in the middle (cf. paper & constraints)
constraint forall(i in 1..n) (
    if thickness[i] mod 2 == 0 then
        table([seq[i, floor(thickness[i]/2)], seq[i, floor(thickness[i]/2)+1]],
            allowed_pair)
    else
        table([seq[i, floor(thickness[i]/2)], seq[i, floor(thickness[i]/2)+1],
            seq[i, floor(thickness[i]/2)+2]], allowed_odd)
    endif
);
% D5 -- surface plies can only take 45/-45 angles
constraint forall(i in 1..n)
    (seq[i, 1] != 0 /\ seq[i, 1] != 2 /\ seq[i, 1] != 4);
constraint forall(i in 1..n)
    (seq[i, thickness[i]] != 0 /\ seq[i, thickness[i]] != 2
        /\ seq[i, thickness[i]] != 4);

% BLENDING CONSTRAINTS
% B1 -- blending rule: every child sequence must be a subset of the parent sequence. For
% every parent node A and child node B: bi = A[indexi]
constraint forall(i in 1..m, j in 1..thickness[edges[i].2])
    (seq[edges[i].2, j] = element(indexes[i, j], seq[edges[i].1, ..]));
% B2 -- no ply-crossing: indexi < indexi+1
constraint forall(i in 1..m, j in 1..thickness[edges[i].2]-1)
    (indexes[i, j] < indexes[i, j+1]);
% B3 -- surface plies must be continuous
constraint forall(i in 1..m)
    (indexes[i, 1] = 1);
constraint forall(i in 1..m)
    (indexes[i, thickness[edges[i].2]] = thickness[edges[i].1]);
% B4 -- max 3 dropped plies: |indexi - indexi+1| <= 4
constraint forall(i in 1..m, j in 1..thickness[edges[i].2]-1)
    ((indexes[i, j+1] - indexes[i, j]) <= 4);
```

Figure 3.5: The complete code of our model in MiniZinc (3/3).

Chapter 4

Search

In this section, we will present the different search strategies that were tested for the composite structure problem. The java code that was used to implement the different search strategies, and everything else related to MaxiCP, can be found here. A search strategy can be divided into two parts:

- **The variable selector** algorithm first selects a variable that will be assigned to a value in the search tree. Many conventional variable selectors exist today, proven to work well on a large set of Constraint Programming problems. However, custom variable selectors, specific to a single problem, can also be implemented. In the following sections, both conventional and custom strategies will be presented.
- **The value selector** algorithm then chooses, once a variable has been selected, what value to assign to that variable. Value selectors are often much simpler than variable selector since they often have less impact on the search. This is also the case for the composite structure problem: we can have hundreds of variables but all those variables have a domain of relatively small size (i.e. the four different angles). In the following sections, a conventional value selector will be presented.

All these search techniques were all developed, and will later in this document be compared using MaxiCP, as MiniZinc does only allow the creation of custom search strategies in a very limited way.

An other thing to understand is the branching scheme we use for our search. While in the branching scheme used in example 1.2.1, a branch was added for every value in the domain of the selected variable, in reality this is not always the way

branching is done. A more popular way of branching, and the branching scheme that will be used here, is to add two branches at each level: one assigns a value to a variable, and the other simply removes that value from the domain of the same variable. As an instance, we can see on the left of figure 4.1 part of the search tree presented in example 1.2.1 while on the right of the same figure we can find the equivalent tree using our branching scheme.

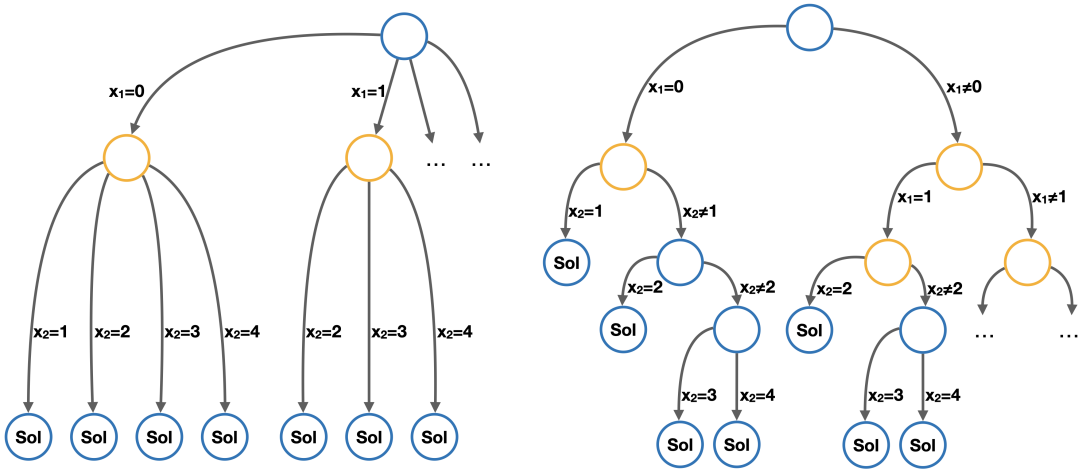

Figure 4.1: On the left, the branching scheme as presented in example 1.2.1, on the right the branching scheme we will use.

While the right search tree seems more complicated (partly because it is), it offers two advantages. The first is that it consumes less memory since it only needs to retain two choices per level instead of a whole bunch. The second advantage comes from the fact that we can make certain decisions later. Indeed, with the image on the left, the order of value assignment is decided at the top, while on the right example, this order can change if wanted.

4.1 Conventional Variable Selectors

In this section, three different well-known conventional variable selectors will be presented.

4.1.1 First Fail

The First Fail strategy [11], as its name suggests, is based on the idea of "failing first" or "failing early". The principle is to select the variable that is most likely to cause a failure as soon as possible. As a quick reminder, a failure (or fail) is when

a value is assigned that violates a constraint of our model. By doing so the search algorithm can quickly backtrack and explore other potential solutions, rather than wasting time on paths that are unlikely to lead to a solution. In other words, it allows to have as much fails as possible happen on top of the search tree, instead of the middle or the bottom of the search tree, which would be detrimental for performance. The first fail strategy therefore systematically selects the variable having the smallest domain at the time of branching.

Example 4.1.1. Consider a simple CSP involving three variables A, B and C with domains $D(A) = \{1, 2\}$, $D(B) = \{1, 2, 3\}$ and $D(C) = \{1, 2, 3, 4\}$. The constraints are such that $A \neq B$, $A \neq C$ and $B \neq C$. In this case, the first fail strategy would first select variable A since it has the smallest domain. It would then try to assign $A = 1$ and proceed to assign values to variables B and C while also respecting the constraints. If a conflict arises, it backtracks and assigns $A = 2$.

By systematically applying this strategy, the algorithm can efficiently explore the solution space and find a valid assignment or determine that no solution exists.

4.1.2 Last Conflict

The last conflict [18] variable selector caches the last variable to cause a fail and prioritizes assignment to that particular variable after backtracking. Since there are no fails at the start of the search, this strategy uses an alternative heuristic until a conflict is found. In our case, this alternative heuristic is the first fail principle.

Example 4.1.2. Consider again a simple CSP involving three variables A, B and C with domains $D(A) = \{1, 2\}$, $D(B) = \{1, 2, 3\}$ and $D(C) = \{1, 2, 3, 4\}$. The constraints are such that $A \neq B$, $A \neq C$ and $B \neq C$. Suppose the algorithm initially assigns $B = 1$ and $C = 2$, but then finds that there is no valid assignment for A anymore. This means that the assignment $C = 2$ caused a fail. The algorithm would then prioritize C for reassigment after backtracking.

4.1.3 Conflict Ordering

The Conflict Ordering [10] variable selector is a generalization of the last conflict strategy. The Conflict Ordering strategy uses information from conflicts encountered during the search to guide the variable ordering, thereby improving the search process. It tries to assign values first to variables that often cause fails and does so by ordering the variables according to a heuristic indicating how likely they are to cause a fail. The variables are then ordered according to the number of times assigning a value to them has caused a fail during the search. Of course, at the

start we do not have any information about which variables are most likely to lead to a conflict. Therefore, at the start we simply adopt the first-fail strategy until we have a fail.

Example 4.1.3. Consider again a simple CSP involving three variables A, B and C. The constraints are such that $A \neq B$, $A \neq C$ and $B \neq C$. Suppose the fail counter of each of the variables is as follows: $counter(A) = 5$, $counter(B) = 2$, $counter(C) = 4$. In this case, variable A will be prioritized for value assignment, then variable C and eventually variable B. Of course, this order can change according to the next fails in the search tree, as the counters are continuously updated.

By dynamically adjusting the variable ordering based on conflict analysis, this strategy can significantly improve the efficiency and effectiveness of solving complex CSPs.

4.2 Custom Variable Selectors

Up until now we have only seen variable selectors that do not take into account the information about the composite structure, such as the thickness of a stacking sequence or how many adjacent sequences a stacking sequence has. In this section, we present variable selectors that try to leverage this information to optimize the search process and minimize the search tree size.

4.2.1 High Degree First

The first custom variable selector that was tested prioritizes stacking sequences that have a high degree in the input graph, thus having many adjacent sequences. The rationale behind this strategy is that sequences with many neighbors influence a greater part of the structures. Thus, by branching on the variables of these stacking sequences first, we have a greater chance of affecting as many variables as possible (since a stacking sequence depends on its neighbors). Additionally, since these variables influence other variables, they are subjected to more constraints and branching on them yields a higher chance of failing.

This reasoning allows us to select a stacking sequence, but we still need to select a particular variable within that stacking sequence. For this, we simply use the same heuristic used for the first fail strategy: we select the variable within the selected stacking sequence that has the smallest domain.

Example 4.2.1. Suppose our input graph is such as the graph presented in figure 4.2. In this case, the first stacking sequence that will be selected for branching during the search will be sequence S_2 since it has three neighbors while all the other only have a single neighbor. Then, the algorithm chooses the variable with the smallest domain and assigns it. When all the variables of this stacking sequence are assigned, the algorithm has to choose the following sequence to branch on. Since they all have the same amount of adjacent sequences, the order will depend on their sequence ID. Thus in this case, once all the variables in S_2 are assigned, the algorithm selects S_1 as next stacking sequence to branch on.

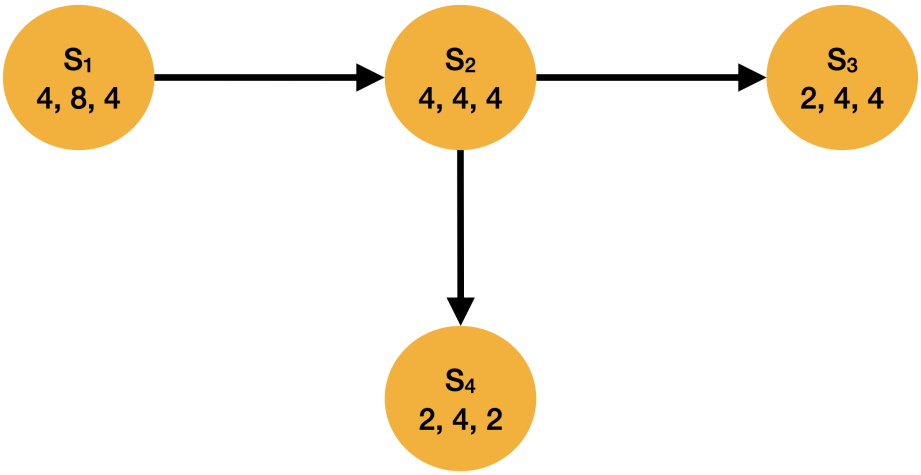

Figure 4.2: An example graph for describing search strategies.

This strategy has the great advantage of being very simple and straightforward to implement. However, since many composite structures are squares and therefore have a lot of stacking sequences with the same amount of neighbors, many decisions are solely based on the ID of the stacking sequence.

4.2.2 Thick Sequences First

As its name suggests, the thick sequences first strategy branches first on the variables of stacking sequences that have a large amount of plies. Once the stacking sequence has been chosen, the algorithm then chooses the variable with the smallest domain.

Example 4.2.2. For instance, on the example graph provided in figure 4.2 the first stacking sequence that would be selected is S_1. Once all the variables in S_1 are assigned, the algorithm chooses S_2, then S_3 and finally S_4. In the event that

two stacking sequences have an identical amount of plies, the sequence ID is again used as a tie breaker.

4.2.3 Thin Sequences First

This is the exact opposite as the previous strategy. Indeed, instead of ordering the stacking sequences from thickest to thinnest we order them from thinnest to thickest. Like all the previous custom search strategies, the algorithm chooses the variable with the smallest domain from the selected stacking sequence.

Example 4.2.3. On the example graph provided in figure 4.2 the first stacking sequence that would be selected is S_4. Once all the variables in S_4 are assigned, the algorithm chooses S_3, then S_2 and finally S_1. In the same way as was done in the previous variable selectors, the sequence ID is used as a tie breaker in the event two stacking sequences have an identical amount of plies.

4.2.4 A Combination of Two Heuristics

One of the problems mentioned in section 4.2.1 is that the variable selector prioritizing stacking sequences with many neighbors, often ends up choosing a stacking sequence based on the sequence ID. This is because many composite structures are rectangular, thus having rectangular stacking sequences which all have four neighbors, except for the ones on the extremities of the structure. One way to improve this strategy is to find a more optimized way of tie breaking the stacking sequences in the event they have the same number of neighbors. This is exactly what we will do with a combination of two heuristics strategy.

In fact, we combine all the custom strategies to create two new ways of branching on the decision variables. Both these techniques use the number of neighbors of a stacking sequence as primary heuristic. The first technique then uses the heuristic presented in section 4.2.2 when two or more sequences have the same number of neighbors, while the second technique uses the heuristic presented in section 4.2.3.

Example 4.2.4. On the example graph provided in figure 4.2 the first stacking sequence selected by both techniques is S_2. Then, the first technique would go on by selecting the thickest sequence, which is S_1, followed by S_3 and finally S_4. The second thechnique would do the opposite after selecting S_2: first S_4, then S_3 and finally S_1.

4.3 A Conventional Value Selector

Since value selectors are not the most influential part in most Constraint Satisfaction Problems, simple algorithms are often used. The conventional value selector we will use and test for this problem will simply assign the variable to the minimum value in the domain of its variable.

Example 4.3.1. Suppose we have a CSP with two variables A and B with domains $D(A) = \{1, 2\}$, $D(B) = \{1, 2, 3\}$ with some constraints. If we use this value selector together with a first fail variable selector, then we would select variable A first and assign the value 1 to it.

Chapter 5

State of the Art

Much previous research has been performed on optimizing composite structures subject to some constraints as can be seen in [26]. In this chapter, we will go over existing solutions for designing composite structures. We will also see what optimization functions are most used for optimizing composite structures.

5.1 Meta-Heuristic Methods

Meta-heuristic algorithms are designed to find near-optimal solutions to complex optimization problems. Their implementation often draws inspiration from natural processes or artificial systems. They are able to navigate complex search spaces and escape local optima, increasing the likelihood of finding a global, or at least a near-global solution. Using meta-heuristic methods is nowadays the most popular way of designing and optimizing composite structures.

For instance, in [9] they use Genetic Algorithms (GA) in order to optimize the buckling load of conventional laminated composite plates. Genetic algorithms are inspired by the process of natural selection: it uses techniques such as selection, crossover and mutation to evolve one or several candidate solutions towards an optimal one. Then, in [16] an improved Ant Colony Optimization (ACO) algorithm is used to reduce the cost of composite beams. Ant colony optimization draws inspiration from the behavior of ants, where artificial ants work together to build solutions by depositing and following a pheromone trail (in our case a heuristic). In [35] Simulated Annealing (SA) is used to maximize the critical buckling load while minimizing weight at the same time for composite beams. Simulated annealing is a probabilistic global optimization technique inspired by the annealing process in

metallurgy, where a material is slowly cooled to achieve a state of minimum energy.

In some cases, hybrid meta-heuristic algorithms are used for designing composite structures. Hybrid meta-heuristic methods combine the strength of two or more individual meta-heurstic methods. They are often used to mitigate the weaknesses of individual meta-heurstic algorithms. For instance, in [21] both genetic algorithms and firefly algorithms, another meta-heuristic method, are used to maximize stiffness while also minimizing weight.

While these methods have proven to work well with this problem, they also have drawbacks. One of the limitations is the parameter tuning required to achieve optimal performance. Indeed, the performance of these algorithms can be very sensitive to the specific values assigned to their control parameters. Finding the best parameters for a given problem requires a lot of time and is a non-trivial task. Furthermore, meta-heuristic methods do not guarantee finding the global optimum of a problem. While they often find a good quality solution, there is no mathematical proof that the best solution can be found. This is one of the major strengths of constraint programming. Indeed, being an exact algorithm it is able to go over the whole search space, and thus find the absolute best solution.

5.2 Exact Methods

A first exact approach was proposed in [43]. It iteratively generates a set of admissible structures (i.e., stacking sequences that meet all coherence constraints), performs numerical simulations, and selects the sequence(s) yielding the highest buckling load. However, it does not include the blending constraints. The approach of Zein et al. (2014) [42] solves the same problem as ours. Despite the title, the problem is not formalized as a declarative CSP. Instead, the authors introduce a custom depth-first search algorithm. During the search, stacking sequences are considered one by one in decreasing order of their number of layers. For each stacking sequence, branching decisions related to blending constraints are made before those related to design constraints. The approach does not rely on a solver with variables, domains, and a fixed-point computation of generic constraints. The algorithm verifies the consistency of the current layer in the stacking sequence with respect to the prefix of already fixed stacking sequences from previous decisions. If an infeasibility is detected for a layer, the search backtracks. The rules are thus enforced as checkers that verify the validity of a prefix, rather than through a filtering algorithm acting on all the variables in its scope as in a standard CP approach. This means that in a stack S of n plies, the ply orientation $S[k]$ is only deemed invalid if it conflicts

with plies $S[1..k-1]$ but no inference is made regarding subsequent plies $S[k+1..n]$.

This algorithm shares similarities with Generative Constraint Programming [34], which lazily prunes only the domain of the next variable to instantiate, taking into account the prefix of already fixed variables. Our approach, on the other hand, relies on a pure declarative model where the search is decoupled from the model. While it is understandable that the custom approach uses a search algorithm that makes decisions in the order of stacking sequences to facilitate the verification of blending constraints. However, it is unclear whether this is the best search strategy, given that all layers are interconnected through blending constraints. Our approach can utilize efficient black-box searches that could detect or learn, that for instance, branching on the thinnest stacking sequence first might be more effective.

5.3 Optimization Functions

Finally, a lot of research has been done on optimizing different objectives and sometimes even a combination of objectives. Maximizing buckling load is one of those objectives. Buckling is a phenomenon where a sudden loss of stability happens in thin, lightweight structures under compressive forces. The buckling load is highly dependent on the stacking sequences and fiber orientations of the plies, which makes it very interesting for our research. Even subtle changes in the stacking sequences can significantly impact the buckling load. However, one of the challenges to this optimization is the computing time needed to calculate the buckling load of a solution. Indeed, as specified in [15], finite element methods are used for calculating the buckling load. Doing this for every solution during the search would cause a great loss of time, and thus alternative ways to estimate the buckling loads are needed.

Another objective that researchers often seek in composite structures is to maximize the natural frequencies of the structure [25]. Maximizing the natural frequencies of a composite structure is important to avoid resonances with external excitations. This ensures dynamic stability and also prevents potential structural damage. This optimization often needs trade-offs with other design objectives such as the weight of a structure. To get a good trade-off, it is often required to use multiple objectives at once. While measuring natural frequencies can be done using finite element analysis, other methods exist. For instance, in [14] the Rayleigh-Ritz method is used, while in [33] they use an artificial neural network to predict the natural frequencies.

Minimizing weight is also a common objective when designing composite structures [28], driven by the need to improve efficiency and reduce material costs while enhancing performance. This is especially the case when weight is an important factor, such as in the aerospace industry. This objective can be achieved through various means: optimizing the ply thickness, choosing the right materials for the structure and determining the most efficient stacking sequence. However, this must be done carefully, since minimizing the weight might have an impact on numerous other characteristics of the structure.

Material strength and failure criteria are also commonly used as objective functions. Implementing these objectives in the program ensures that the structure withstands the expected loads without experiencing a failure. Various failure criteria exist, such as the Tsai-Wu [19] criterion or the maximum stress-strain criteria [1].

Finally, many other common objectives exist, such as cost minimization, deflection minimization or even the use of constraints as an objective. Many methods try to optimize several objectives at once, or try to optimize an objective subject to another objective. While optimizing the structures is interesting, there are major challenges to overcome such as the computation time for some of the objectives.

Chapter 6

Experiments

To assess the performance of our model and search strategies, we generated in agreement with our industrial partner Cenaero, a set of instances representative of the ones that are generally processed by their software. Four datasets were generated using grid sizes of 3x3, 4x4, 5x5 and 6x6, stacking sequences, each containing 40 instances. Each node in the grid is connected to a minimum of two and up to four of its neighbors (top, left, bottom, right) in the grid by a directed edge, whose direction is chosen randomly. The cardinality constraints on the number of plies per orientation in each stacking sequence were selected uniformly with up to 60 plies per stacking sequence. The retained instances are such that none is trivially detected as infeasible by the industrial solver when adding all constraints (B1-B4, D1-D5).

6.1 D2 & D3 Constrain Formulation

As mentioned in section 3.1.3, different constraint formulation can have a significant performance impact on the model. In this section, we study what the best constraint formulation is for the D2 and the D3 rules. Three different configurations were benchmarked for these two rules:

1. The first one enforces the D2 and the D3 rule in their simplest form, without using table constraints.
2. The second configuration enforces the D2 rule as described in its simplest form and enforces the D3 rule using a table constraint.
3. The third configuration enforces both the D2 and the D3 rule using a single table constraint.

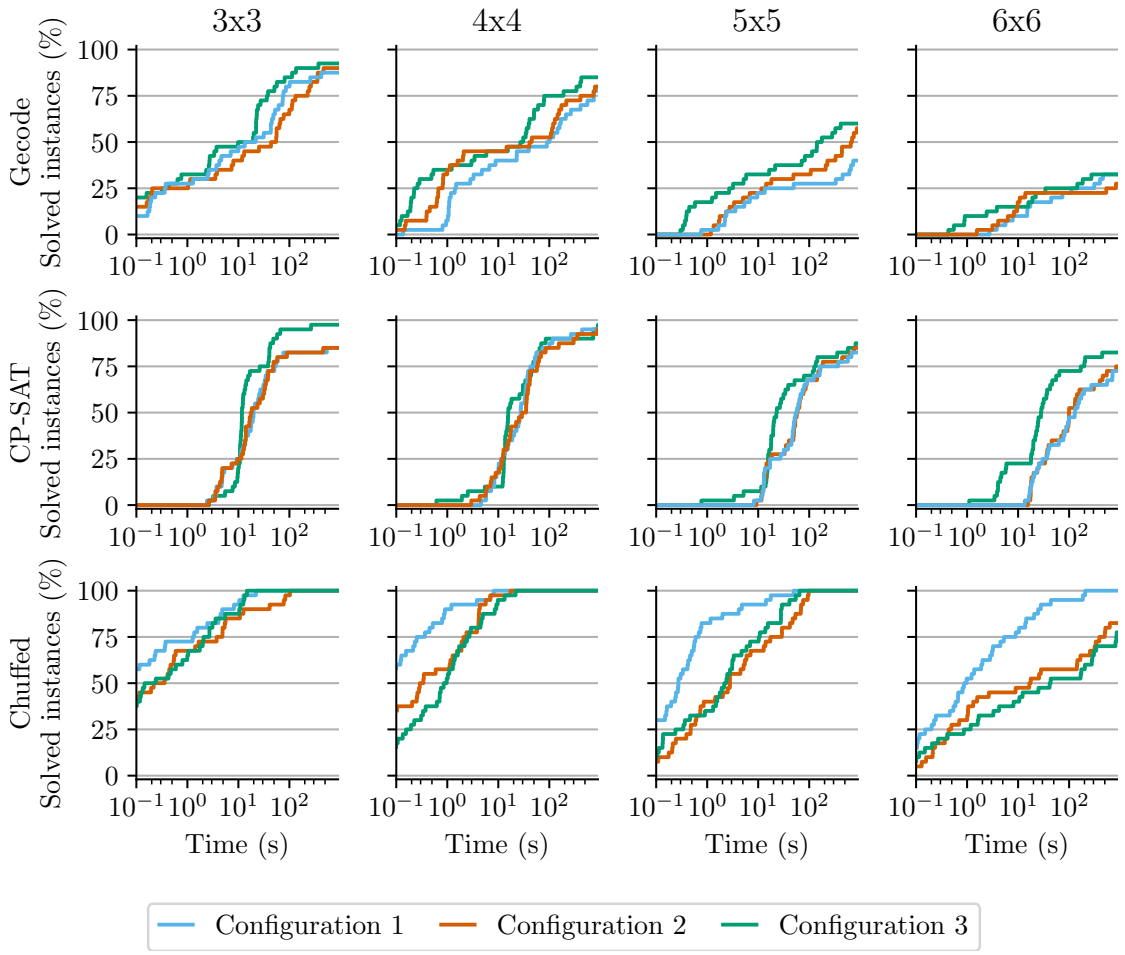

Figure 6.1: The three configurations for enforcing the D2 and the D3 rule, benchmarked against several test cases.

These three constraints were tested on a set of benchmark instances, the result of which is presented in figure 6.1. In this figure, we can see an array of plots. Each row represents a solver, while each column represents a different type of benchmarks. The solvers used are: the CP-SAT solver [31, 6, 30, 32], the Gecode solver [38] and the Chuffed solver [5]. There are four different types of benchmark instances using grid sizes of 3x3, 4x4, 5x5 and 6x6 stacking sequences. On every plot, the three different configurations are compared against each other. Every plot represents the percentage of instances solved over time using a timeout of 15 minutes, meaning that if we do not find a solution after 15 minutes, we forcefully stop the search.

On the results presented in figure 6.1, we can firstly see that the Gecode solvers's performance improves slightly going from configuration 1 to configuration 2 and then improves slightly again going from configuration 2 to configuration 3, indicating that for Gecode configuration 3 is the most efficient on these set of benchmarks. However, neither of the results are great for the Gecode solver. A larger difference can be seen using the CP-SAT solver, which follows the same trend but performs better overall. Indeed, while its performance does not seem to change going from configuration 1 to configuration 2, it significantly improves going from configuration 2 to configuration 3. After further analysis, it was apparent that this was mainly due to the instances where the constraints could not be satisfied, and thus no solution was found (also called "unsatisfiable" or "UNSAT" instances as opposed to "SAT" instances for solvable instances). This can be seen in figure 6.2, where on the left we can see the average number of failures and the average time taken to solve the SAT instances and on the right we can see the same for the "UNSAT" instances. Both graphs were made using the CP-SAT solver and were measured only on the instances where all three configurations found a solution in less than 15 minutes.

In this figure, we can first see that configuration 2 slightly improves over configuration 1 in both time and failures, for "SAT" and "UNSAT" instances. These changes are, however, relatively small. Then, for configuration 3, using only the table constraint for the D2 and the D3 constraint, we can see for the "SAT" instances that it reduces failures, meaning that it might have a stronger filtering algorithm. However, this decline in failures does not translate in a decrease in the time taken to find the first solution. Indeed, the time taken to find a first solution for the "SAT" instances almost doubles. This is likely due to the fact that, while the filtering algorithm is stronger, it takes more time to filter and thus the benefits of the more effective filtering disappear. This is not the case for the "UNSAT" instances, where we can see a significant decrease in both failures and time to find the first solution. It is clear that for those instances, having a stronger filtering algorithm with CP-SAT presents great benefits.

The Chuffed solver breaks this trend by notably performing better on configuration 1 compared to the other configurations. This time, if we look at number of failures and the time taken to solve the problem in figure 6.3, we can see that the time and failure gain comes mainly from solving the SAT instances. Indeed, configuration 3 performs more than ten times worse compared to configuration 1 both in terms of failures as in terms of time. For the UNSAT instances on the right, we can see that the average number of failures decreases slightly when using configuration

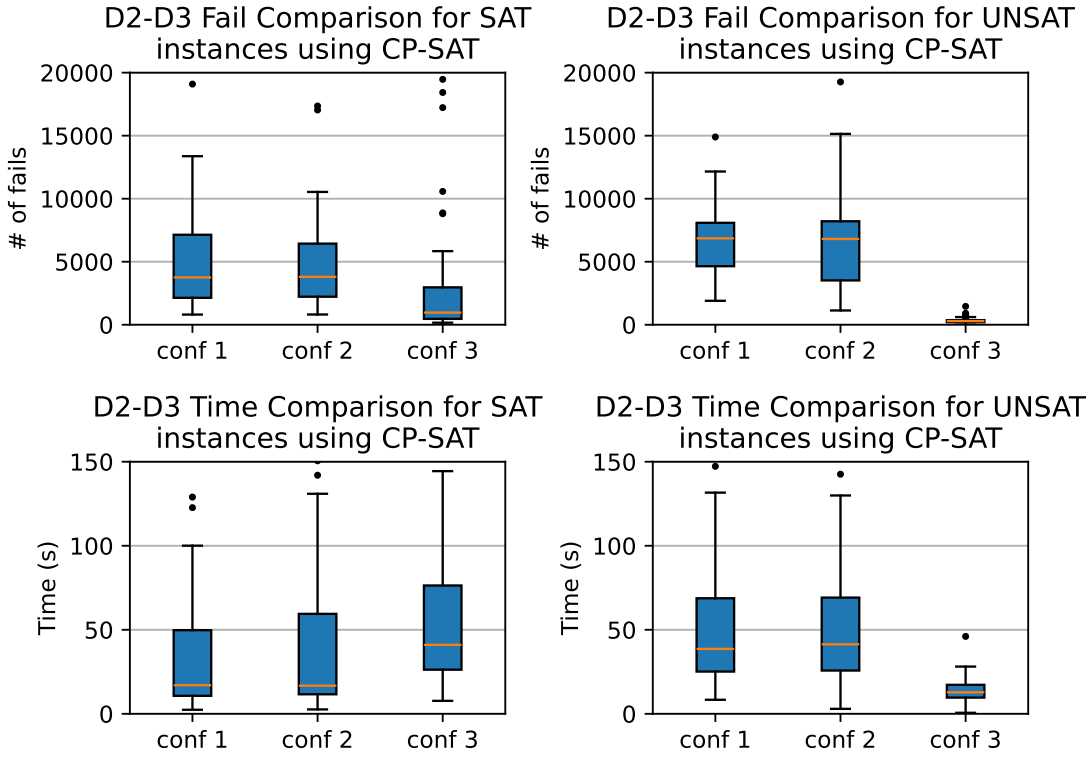

Figure 6.2: Box plot showing the failures and the time taken to solve a composite structure instance, using different constraint formulation on the CP-SAT solver.

3, but this change cannot be seen when looking at the time, which remains more or less the same. Following these results, the Gecode and CP-SAT solvers will use configuration 3 for the remainder of the benchmarks in this thesis, while the Chuffed solver will use configuration 1.

6.2 Solvers

For comparing solver performance on our MiniZinc model, the same three solvers as used in section 3.1.3 were used again: CP-SAT, Chuffed and Gecode. Two Mixed Integer Linear Programming (MILP) [41] solvers using their MiniZinc backend were also tested: Scip Optimization suite [2, 3] and HiGHS [13]. Lastly, the performance of the solution deployed by our industrial partner Cenaero, that implements the approach of [42] with all constraints activated is also reported. The code for all the tested models, except for the industrial solution, can be found here.

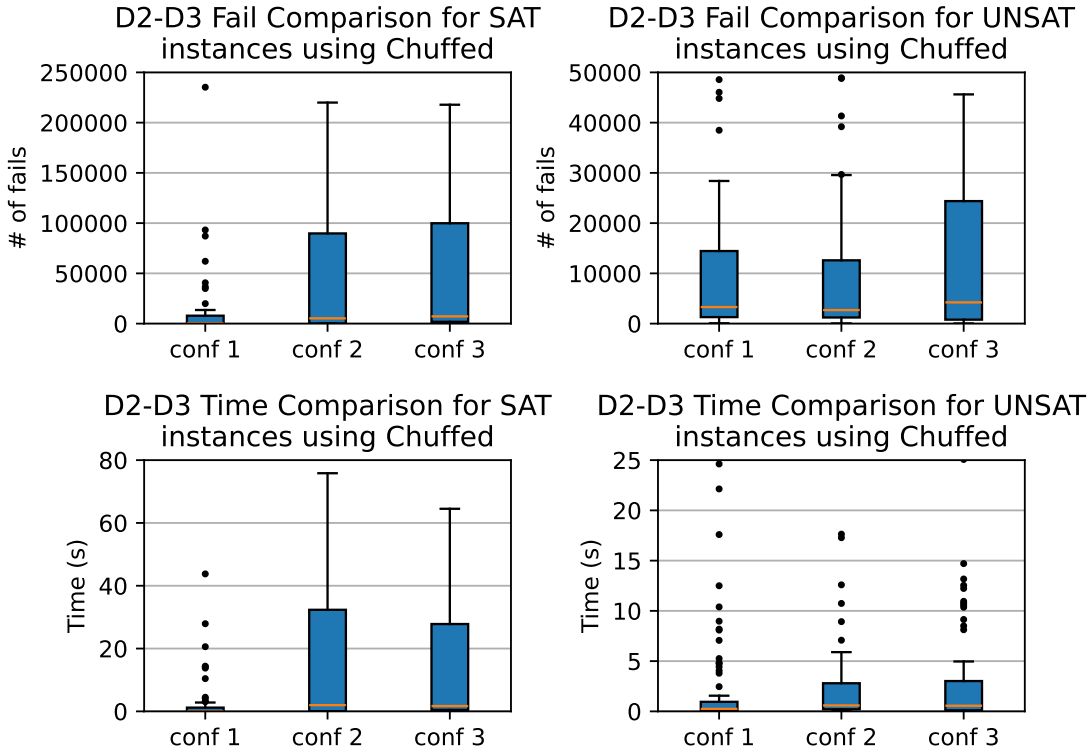

Figure 6.3: Box plot showing the failures and the time taken to solve a composite structure instance, using different constraint formulation on the Chuffed solver.

Figure 6.4 reports the number of instances solved over time with the three tested models, using a timeout of 15 minutes. Each model always includes constraints B1-B4 and enforces a minimum of 8% of every angle in every stacking sequence even when the D1 constraint is deactivated. The first model (top) does not include any optional constraints, the second (middle-top) adds constraints D1-D3 and D5, the third (middle-bottom) adds constraints D3 and D4 and the fourth (bottom) includes all constraints.

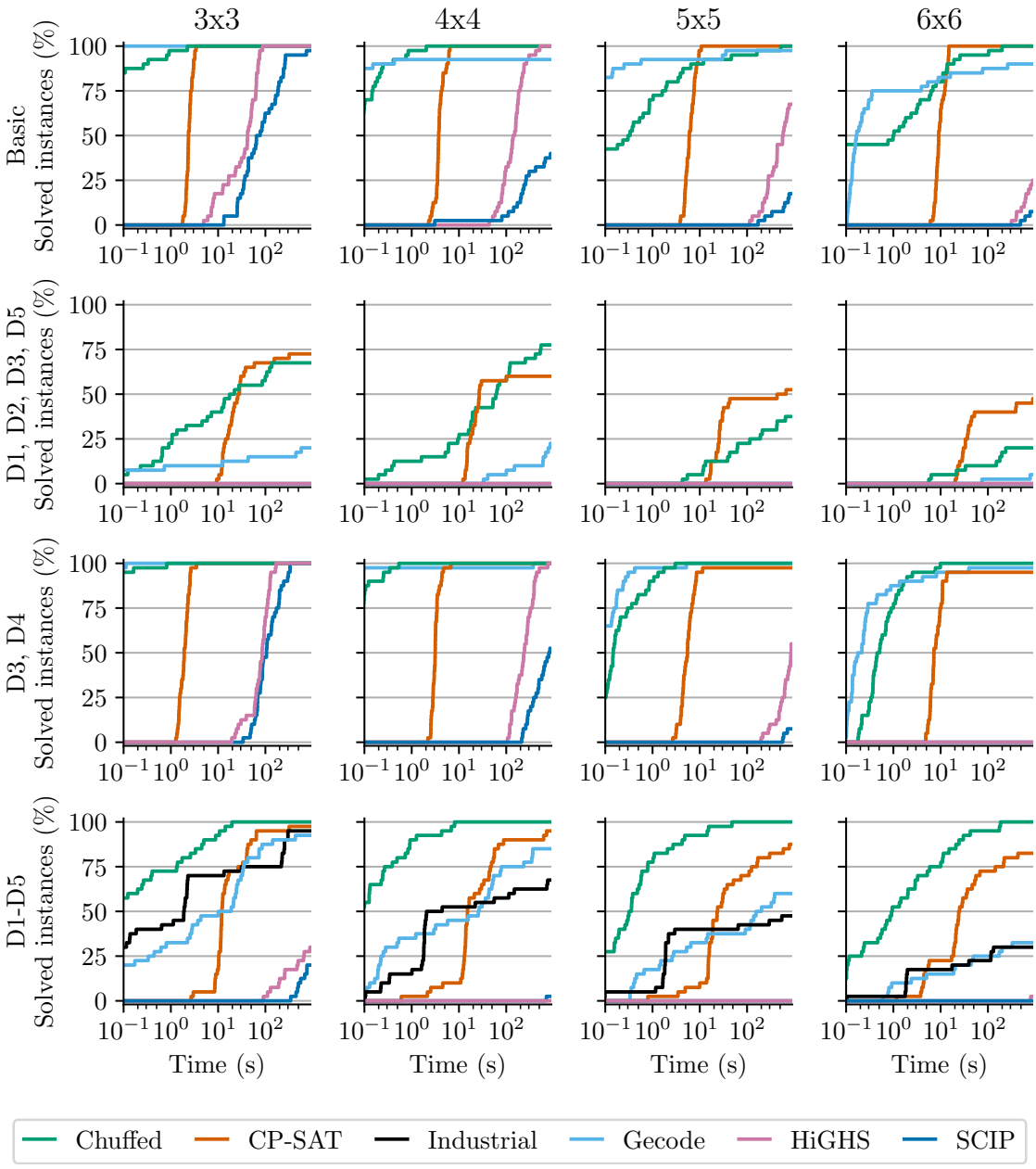

Figure 6.4: Solver performances with different sets of design constraints (rows). Each column denotes a given dataset.

Chuffed appears as the most stable, and overall, best solver, consistently solving 100% of the instances within 15 minutes in all datasets and all models but the second one. The strong point of CP-SAT is that its performance does not degrade

much when the size of the instances increases, which is not the case with Gecode, Chuffed and the industrial solver. Although not visible in this aggregated plot, Gecode is generally faster on feasible instances, while CP-SAT performs better at proving infeasibility. MILP solvers manage to solve some instances, in particular when not too many constraints ae involved and on smaller datasets. However, they are both marginally behind the other solvers and even behind the industrial solution. Moreover, Highs performs better than Scip. The industrial solver achieves performance competitive with Gecode but is outperformed by the lazy-clause generation solvers [27] CP-SAT and Chuffed. This indicates that clause learning significantly enhances the ability to solve this problem robustly.

On the second model, we can see how the different models perform with the D4, symmetry constraint, deactivated. It becomes clear that without this constraint, the problem becomes much harder. This is because the symmetry constraint allows MiniZinc to reduce the number of variables, and thus the magnitude of the problem, to about half of them. It also shows the difficulty to satisfy the cardinality constraint in particular. This particular model also shows once again the superiority of CP-SAT in finding unfeasible instances. Indeed, while not shown here, almost all the solved instances were unfeasible, for all the solvers with CP-SAT once again the most consistent solver.

6.3 Variable Selectors

In figure 6.5 we can see the performance of the conventional variable selectors, presented in section 4.1, on a MaxiCP model. The MaxiCP model used here has all the optional constraints (D1-D5) enabled.

The conventional variable selectors perform very similarly amongst them. The conlfict ordering strategy performs best on the three smaller datasets (3x3, 4x4 and 5x5) but is then outperformed by the first fail strategy on the 6x6 dataset. Interestingly, the first fail strategy and the last conflict strategy perform the worst on the 4x4 dataset and then improve on the 5x5 and 6x6 datasets. In the end, none of the conventional variable selectors perform great, especially on the larger datasets.

If we compare the conventional variable selectors to our custom variable selectors, benchmarked in figure 6.6 and previously presented in section 4.2, this time some differences are visible. First of all, all of the custom variable selectors improve significantly for the 3x3 and the 4x4 datasets over the conventional variable selec-

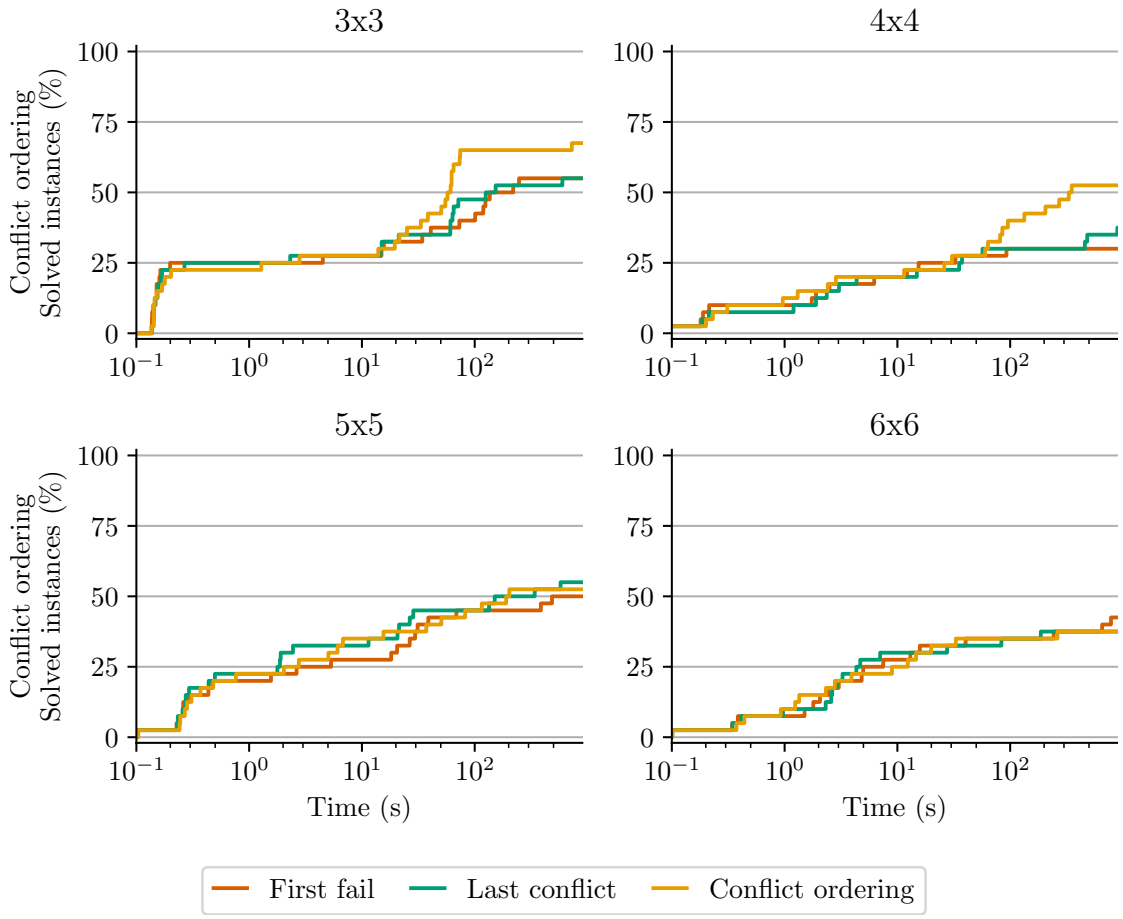

Figure 6.5: Solver performance with different conventional variable selectors (rows). Each column denotes a given dataset.

tors. Then, for the 5x5 dataset, the High thin first strategy leads the pack with a significant advantage over the conventional strategies, the Thin first strategy performs similar to the conventional ones, while the Thick first strategy underperforms compared to the conventional strategies, indicating that sorting the stacking sequences from thick to thin might not be the way to go. For the 6x6 dataset, the story changes a bit. While the Thick first strategy is still worse than the convenitional strategies, the Thin first strategy is better than them on this dataset. Another difference on this dataset is that the High thick first strategy does not outperform the conventional variable selectors but instead performs similarly, indicating once again that sorting from thick to thin might not be the best idea here.

If we now compare the custom strategies amongst them we see that the story remains the same. The strategies that sort the sequences from thick to thin (i.e. the Thick first and the High thick first strategies) perform the worst while the Thin first strategy performs better than the conventional ones, but does not manage to outperform the High degree and High Thin first strategies. Looking at the performance of those two strategies tells us that starting at the stacking sequence with many adjacent sequences is indeed the way to go. However, it must be noted that using the Thin first ordering as opposed to simply ordering based on the sequence ID does not yield very significant performance increases. In fact, even though the High thin first performs better on the 5x5 and 6x6 datasets, it performs the same at best for the 3x3 and 4x4 datasets. This might have two explanations.

The first explanations is simply that the thickness of the stacking sequence might not be such a great heuristic to select our variable. However, we have seen that the Thin first strategy performs at least similarly to the conventional stragies, indicating that it is not such a bad heuristic either. The second explanation involves the way that the vanilla High degree strategy orders the stacking sequences when there are ties. It was previously said (in section 4.2.1) that if there are two stacking sequences having the same number of edges, the stacking sequence ID would be used to break ties. Since many stacking sequences have the same degree, this tie breaker is used a lot and thus a large part of the stacking sequences are ordered according to their ID. Now, something to know is that two IDs that are close to each other (say stacking sequence 1 and stacking sequence 2) are often neighbors (i.e. meaning that those stacking sequences are adjacent to one another). Fixing a whole stacking sequence has a great impact on the plies of its neighboring stacking sequences, thus, following the first fail principle, it is interesting to then fix those neighboring plies first, as their domain is greatly reduced. Therefore, using the stacking sequence ID as a tie breaker turned out to be not such a bad idea.

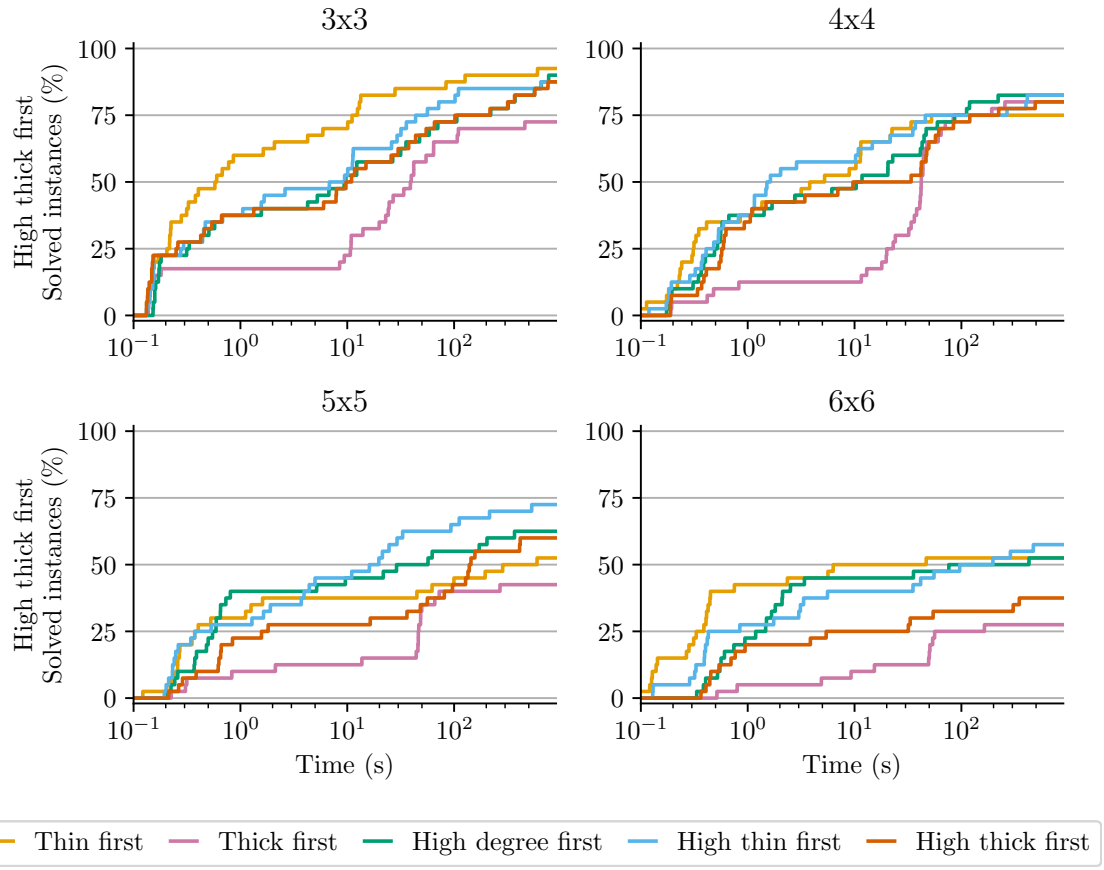

Figure 6.6: Solver performance with different custom variable selectors (rows). Each column denotes a given dataset.

Chapter 7

Solving Unfeasible Instances

One interesting question one might ask about this problem is how to solve unfeasible instances. Indeed, while an instance might be unfeasible with all the optional constraints activated, we might be able to find a compromise, providing an approximate solution that violates as few constraints as possible. In this chapter, we will look at what constraints are the hardest to satisfy and search for ways to relax them while still aiming to enforce them as much as possible. The code of the models that will be preseneted in this chapter can be found here.

7.1 Constraint Analysis

First, we will look at the impact the optional constraints on the feasibility of the problem. This is done by taking away one specific constraint from the model and then testing how many unfeasible instances it finds out of all the benchmark instances. We did this for the D1, D2, D3 and the D4 constraints but not the D5 constraint as its impact is limited. The results of this are shown in figure 7.1, where we can see in blue the percentage of SAT instances for each model, in orange the percentage of UNSAT instances for each model and in green the percentage of instances that could not be solved in less than 15 minutes by any solver. The models shown on the x axis are the following:

- **Basic** represents the model with all optional constraints activated. It will serve as reference in this comparison.
- **90 Gap Allowed** removes the D2 constraint that prohibits angle differences of 90° in between two consecutive plies.
- **Max 4 Consecutive** relaxes the D3 constraint to allow up to four consecutive

plies of the same angle instead of only three.

- **Max 5 Consecutive** relaxes the D3 constraint to allow up to five consecutive plies of the same angle instead of only three.
- **Dissymmetry Allowed** removes the D4 constraint that forces a symmetry between the upper half and the lower half of every stacking sequence.
- **No Cardinalities** removes the D1 constraint that forces a given amount of plies of every angle in every stacking sequence. Note that while this constraint is removed, the model still enforces a minimum of 8% of plies of every angle in every stacking sequence.

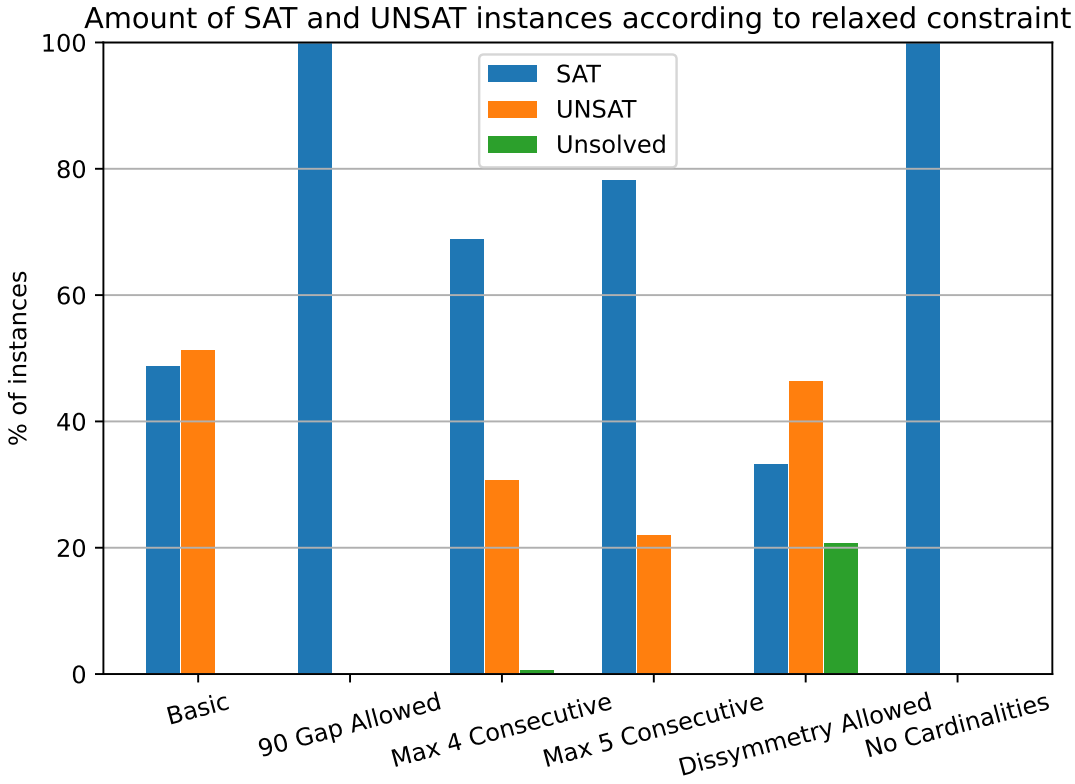

Figure 7.1: Comparison of the amount of SAT and UNSAT instances according to which optional constraint is removed.

In figure 7.1 the first thing we see is that on the Basic model, there are about half of the instances that are feasible and the other half is unfeasible. We are looking to maximize the number of feasible instances by removing an optional constraint. The two models that stand out are the **90 Gap Allowed** and the **No Cardinalities** models, indicating that the D1 and D2 constraints are the hardest to satisfy. Indeed,

when removing one of these two constraints, all the instances become easily feasible and, although no numbers are shown here, they are resolved much faster. Then, the **Max 4 Consecutive** and **Max 5 Consecutive** models increase the amount of feasible instances but do not manage to find a feasible solution for all the instances. Finally, the **Dissymmetry Allowed** does not improve the number of feasible solutions by a significant amount and, problematically, makes the problem harder to solve with a significant increase in the number of unsolved instances. Removing the D4 constraint from our model is thus not a great idea.

7.2 Translating a Constraint into an Objective

When removing a constraint, we can see that we can potentially make all of the instances feasible. Ideally, we would still try to respect the constraint as much as possible and only relax it where needed to make the instance feasible. One way to do this is to translate the constraint into an objective. Once the constraint is translated into an objective, we can try to minimize it when searching for a solution. Instead of only searching a single solution, we will find a solution and then try to find subsequent solutions that improve this objective, and by doing so, partly implement the removed constraint. The best solution in this case is the one that improves the objective the best.

In figure 7.2 we can see the objective function that can be used for the model where the D2 constraint is removed. On line 2, of this figure, the places where a 90° difference between two plies is present are stored in boolean variables, with the value `true` meaning that a 90° gap is present and `false` meaning that it is absent. Then at line 6 we count the number of 90° gaps there are in all the stacking sequences and store this number in an integer variable. Finally, the `solve minimize objective` statement means that during the search we will try to minimize the amount of 90° gaps present in the stacking sequences.

```
1 % Store in boolean variables if there is a 90 degree gap between two consecutive plies in
2 % the stacking sequence.
3 array[1..n, 1..T-1] of var bool: gap90_presence = array2d(1..n, 1..T-1,
4     [abs(seq[i, j] - seq[i, j+1]) == 2 | i in 1..n, j in 1..T-1]);
5 % Sum the number of 90 degree gaps for each stacking sequence.
6 var int: objective = sum(i in 1..n, j in 1..thickness[i]-1) (gap90_presence[i, j]);
7 solve minimize objective;
```

Figure 7.2: Objective function to decrease the amount of 90° angle differences in between two consecutive plies.

Then, in figure 7.3 the objective function used for penalizing sequences of plies containing more than three consecutive plies with the same angle is shown. Indeed, first we store in a boolean array the occurrences of four or more consecutive plies having the same angle and then we count how many times this occurs and store it in the `objective` variable. Finally at line 8, we indicate that we want to minimize this number of occurrences. This objective can be used when removing or relaxing the D3 constraint.

```
1 % Store in boolean variables where there are 4 consecutive plies in the same direction
2 % in the stacking sequence.
3 array[1..n, 1..T-3] of var bool: cons4_presence = array2d(1..n, 1..T-3,
4     [seq[i, j] == seq[i, j+1] /\ seq[i, j] == seq[i, j+2] /\ seq[i, j] == seq[i, j+3]
5     | i in 1..n, j in 1..T-3]);
6 % Sum the number of 4 consecutive plies for each stacking sequence.
7 var int: objective = sum(i in 1..n, j in 1..thickness[i]-3) (cons4_presence[i, j]);
8 solve minimize objective;
```

Figure 7.3: Objective function to decrease the amount of four consecutive plies with the same angle.

In figure 7.4 we can see the objective function for when the D1 constraint is removed from the model. We first count the amount of plies of each angle that are present in every stacking sequence and store these amounts in an array of integer variables. Then, we calculate the difference with the expected number of plies for every angle and try to minimize these differences.

```
1  % Store the number of plies in each direction for each stacking sequence.
2  array[1..n, 1..4] of var 0..T: ply_count = array2d(1..n, 1..4, [count(seq[i, ..], j)
3      | i in 1..n, j in 1..4]);
4  % Sum up the difference between the number of plies in each direction and the expected
5  % number of plies.
6  var int: objective = sum(i in 1..n, j in 1..4) (
7      if j == 1 \/ j == 3 then
8          abs(ply_count[i, j] - ceil((counts[i, 1] + counts[i, 3]) / 2))
9      else
10         abs(ply_count[i, j] - counts[i, j])
11     endif
12 );
13 solve minimize objective;
```

Figure 7.4: Objective function to respect the given cardinalities as much as possible.

7.2.1 Results Using Objective Functions

The objective functions presented in the previous section will be tested here and their results will be discrussed. In figure 7.5 we can see the value of the objective

functions found on the UNSAT instances after 30 minutes of running time. All the results were generated using the Chuffed solver.

We can see that the model that removes the D2 constraint manages to find solutions with very few 90° gaps, especially for the small instances. The two models that relax the D3 constraint are also able to minimize the number of violations on the bigger instances. The model that removes the D1 constraint seems to be the hardest to find a good solution on. Overall, there does not seem to be a single objective that performs great on all the UNSAT instances, indicating that using a combination of relaxed constraint might lead to better solutions.

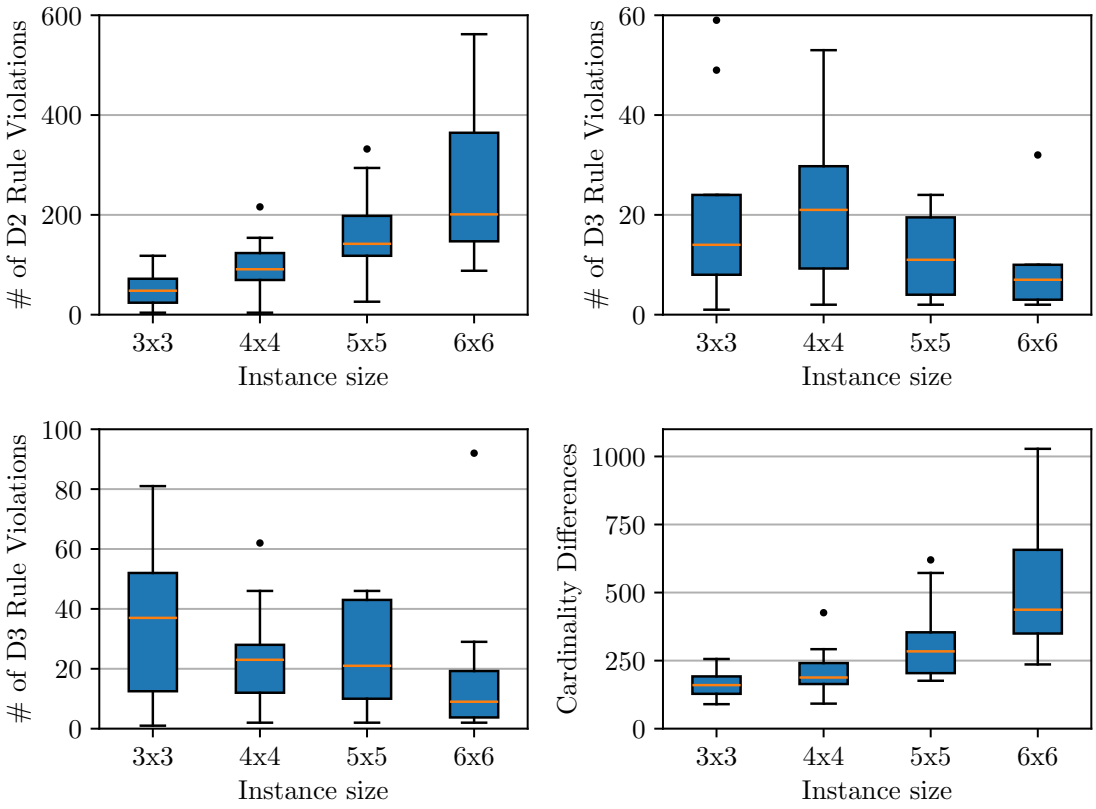

Figure 7.5: Value of the objective function after 30 minutes of running time on UNSAT instances using (from left to right, top to bottom) the **90 Gap Allowed**, the **Max 4 Consecutive**, the **Max 5 Consecutive** and the **No Cardinalities** models.

Chapter 8

Future Works

Since designing composite structures is a new problem for Constraint Programming, a lot of ways exist that might improve the performance of the solvers on this particular problem. In this section, ways to potentially improve the design of composite structures using Constraint Programming will be presented. These techniques aim to both improve the time to find a solution as well as the quality of the solution.

8.1 Optimization Function

While this thesis focused on finding a first solution to the design problem, ultimately the goal is to find the best composite structure design subject to some, or multiple objective functions. A couple of such objectives were presented in section 5.3. There are two main obstacles to adding optimization functions. The first one is that the search must be optimized enough to find a first, and subsequent, solutions. Otherwise, the objective function will serve to nothing. In this thesis it was shown that finding the first solution is possible within reasonable time. There is however still a lot of room for optimization and improvement. Then, the second obstacle is that many of those objective functions need a finite element simulator that allows for a more fine-grained estimation of the mechanical properties of the composite structure.

In future works, it might be interesting to include some possible objective functions or hybridize the model with such a finite element simulator that would allow for finding a composite structure design that improves some aspects. An interesting approach we could try in that direction is the so-called *Empirical Decision Model*

Learning framework [20].

8.2 Redundant Constraints

Another area where this problem could possibly be improved is in the filtering. Improving the filtering of the domains can be done by adding so-called redundant constraints. A redundant constraint does not exclude any solution that was previously valid, but enhances the pruning of the search space by improving the communication between different constraints. It can for instance combine two existing constraints into one to filter values at an earlier stage in the search tree.

An example of such a redundant constraint for this problem is the AtMostSeqCard constraint from [40]. This constraint limits the number of consecutive variables that can take a specific value v and restricts the total number of variables instantiated to v. In other words, the AtMostSeqCard constraint is a combination of the D1 cardinality constraint and the D3 constraint disallowing four or more consecutive plies with the same angle. Unfortunately, the standard MiniZinc library does not include it, hence it has to be added directly to the solver, without going through MiniZinc.

While we cannot test the performance of adding this constraint in terms of the amount of instances solved within a given time without implementing it manually in a CP solver, we can still test its potential by simulating the constraint. This is indeed what we did, using the small input graph shown in figure 8.1. To simulate the atMostSeqCard constraint, we used a table constraint and allowed the same tuples that would be allowed by the atMostSeqCard constraint, the code of this model can be found here. This will not give us any indication in terms of the time to solve an instance, but it gives us an idea of the impact that the constraint can have in terms of failures during the search. This is exactly what is shown in figure 8.2, where we can see the amount of failures on the small instance in figure 8.1 using two different models. The bars in blue represent a normal model with all the optional constraints activated, the orange bars represent the same model, but with the added, simulated, atMostSeqCard constraint.

It is clear that for most solvers, except CP-SAT, a slight to medium improvement can be noted. However, since the instance used here is quiet small, it is harder to get four or more consecutive plies with the same angle. Indeed, while we cannot know for sure, it is likely that the gain in failures improves on larger instances, as

there is a bigger probability of having more than three consecutive plies with the same angle, since the cardinalities are higher. The reason that a larger instance was not used here is because it is hard to generate all allowed tuples to simulate the atMostSeqCard constraint.

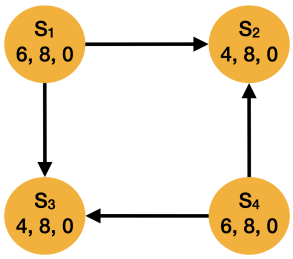

Figure 8.1: Small input graph used for testing the potential performance impact of the atMostSeqCard constraint.

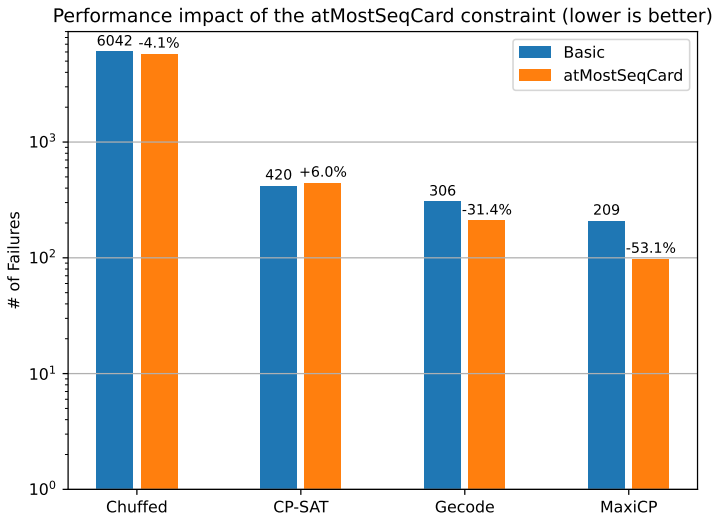

Figure 8.2: Evolution of the number of failures on a small instance using a model with all constraints activated, with and without the atMostSeqCard constraint.

8.3 Search Strategies

While a couple of custom search strategies were already presented in section 4.2 and then tested in section 6.3, there is always room for improvement. For instance, when testing the custom variable selectors we found that ordering on the stacking sequence ID actually worked well. It might be interesting to find the cause to this and exploit this to extract even more performance from the search.

Additionally, we can try to combine one of our custom search strategy with conflict-based searches [10] and restart-based strategies with no-good learning [18]. Once an objective function is implemented, we can also try to enhance our search using Large Neighborhood Search (LNS) [39]. LNS tries to improve an existing solution by altering it using a heuristic. If we let the normal search run on one thread, and then let LNS run on another thread, trying to find improved solutions based on the ones found by the normal search, we could greatly reduce our total search space.

8.4 Alternative Formulations

In this thesis, the model is pretty straightforward, having one variable with an integer domain for every ply in every stacking sequence. However, alternative CP formulations using, for instance, sequence variables [7] could be used to create an alternative formulation of the problem and potentially find new ways to enhance the performance. Another example is using boolean variables to define the ply drop-offs in between stacking sequences as was done in [43]. Many more implementation techniques exist and may be explored. Different techniques can impact the performance of a model greatly as it changes the way how variables are linked to each other and to the constraints.

8.5 Solving Unfeasible Instances

While we studied the impact of the different optional constraints on unfeasible problems and tried to translate some of them into objective functions to respect them as much as possible without enforcing them, many ways can still be found to improve this problem. Indeed, different constraints could be merged into a single objective function for instance, trying to find a compromise between different constraints instead of only focusing on a single constraint. We could also try to relax some constraints, instead of completely deleting them, for instance, we could enforce the D1 constraint only partly, allowing a little bit of slack on the cardinalities. We could also combine those two methods. Many more ways can be found to improve this aspect of the problem and can be explored in future works.

8.6 Adding Other Constraints

While the constraints used in this thesis are the ones that are often found in other research papers as well, it is not unthinkable that the customer would want a different constraint. An example of another constraint is the R5 constraint cited in [42]. This constraint enforces a uniform distribution of the 0° and 90° plies across the whole stacking sequence. We did not implement it since we could not find a formal and detailed definition of this constraint, but it would be interesting to add such constraints in the future and see how they impact the problem.

8.7 Interface

At this moment, the solution exists only in a MiniZinc model. Once that the solution has be refined further, one could imagine implementing a user-friendly interface

to make or load an input graph, launch the program and get the results back. We could, for example, also add useful feedback for the user when encountering unfeasible solutions using Large Language Models. It could also be interesting to adopt the standards of the industry for the input files and generating the output of the program. This way the user can immediately interpret and use the solution using its own tools, without needing to convert the solution.

Chapter 9

Conclusion

Designing composite structures is combinatorial problem because of the many variables and constraints involved. Today, meta-heuristic algorithms are primarily used to satisfy theses constraints. In this thesis, we tried to improve the design process by introducing a new way to design composite structures. A model using Constraint Programming was shown and the constraints required for such structures were formally defined. Then, a model was implemented with MiniZinc and tested using off the shelf solvers such as CP-SAT, Gecode and Chuffed. The model implemented in MiniZinc significantly improved over the state of the art solver of our industrial partner on a large set of different benchmark instances, showing the potential of constraint programming on this problem. Afterwards, both conventional and custom searches were tested on this problem using MaxiCP, with varying results. However, in the end, one particular search strategy was shown to be superior over the others. Another aspect of this problem that was presented is how do we find a solution to instances that are infeasible when all the constraints are activated. We analyzed which constraint are the hardest to satisfy and tried to translate them into objectives to find a solution that compromizes a constraint in favor of finding a feasible solution. Finally, different ways were presented to go further on this topic and to optimize the model and searches developed during this thesis.
We can conclude that Constraint Programming shows a lot of potential for designing composite structures. However, there is still a lot of space for improvement of our model as discussed in chapter 8.

The work presented in this thesis has also been published in the following research paper: Antoons Miguel, Delecluse Augustin, Schaus Pierre, and Zein Samih. "Modeling and Solving a Composite Structure Design Problem with Constraint Programming". In: *CP2025* (2025).

Bibliography

[1] Sadao Amijima and Tsuneyuki Adachi. "Nonlinear stress-strain response of laminated composites". In: *Journal of composite materials* 13.3 (1979), pp. 206–218.

[2] Suresh Bolusani et al. *The SCIP Optimization Suite 9.0*. Technical Report. Optimization Online, 2024. URL: https://optimization-online.org/2024/02/the-scip-optimization-suite-9-0/.

[3] Suresh Bolusani et al. *The SCIP Optimization Suite 9.0*. ZIB-Report 24-02-29. Zuse Institute Berlin, 2024. URL: https://nbn-resolving.org/urn:nbn:de:0297-zib-95528.

[4] Kenil CK Cheng and Roland HC Yap. "An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints". In: *Constraints* 15.2 (2010), pp. 265–304.

[5] Geoffrey Chu et al. *Chuffed - A lazy clause solver*. 2016. URL: https://github.com/chuffed/chuffed.

[6] CPAIOR. *CPAIOR 2020 Master Class: Constraint Programming*. 2020. URL: https://youtu.be/lmy1ddn4cyw.

[7] Augustin Delecluse, Pierre Schaus, and Pascal Van Hentenryck. "Sequence Variables for Routing Problems". In: *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Ed. by Christine Solnon. Vol. 235. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 19:1–19:17. ISBN: 978-3-95977-240-2. DOI: 10.4230/LIPIcs.CP.2022.19. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2022.19.

[8] Jordan Demeulenaere et al. "Compact-table: efficiently filtering table constraints with reversible sparse bit-sets". In: *Principles and Practice of Constraint Programming: 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22*. Springer. 2016, pp. 207–223.

[9] H Arda Deveci, Levent Aydin, and H Seçil Artem. "Buckling optimization of composite laminates using a hybrid algorithm under Puck failure criterion constraint". In: *Journal of Reinforced Plastics and Composites* 35.16 (2016),

pp. 1233–1247. DOI: 10.1177/0731684416646860. eprint: https://doi.org/10.1177/0731684416646860. URL: https://doi.org/10.1177/0731684416646860.

[10] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. "Conflict ordering search for scheduling problems". In: *Principles and Practice of Constraint Programming: 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*. Springer. 2015, pp. 140–148.

[11] Robert M. Haralick and Gordon L. Elliott. "Increasing tree search efficiency for constraint satisfaction problems". In: *Artificial Intelligence* 14.3 (1980), pp. 263–313. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(80)90051-X. URL: https://www.sciencedirect.com/science/article/pii/000437028090051X.

[12] Pascal Van Hentenryck and Jean-Philippe Carillon. "Generality versus specificity: An experience with AI and OR techniques". In: *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence*. 1988, pp. 660–664.

[13] Qi Huangfu and JA Julian Hall. "Parallelizing the dual revised simplex method". In: *Mathematical Programming Computation* 10.1 (2018), pp. 119–142.

[14] R Jafari, P Yousefi, and Sh Hosseini-Hashemi. "Vibration optimization of skew composite plates using the Rayleigh-Ritz & response surface methods". In: *International conference on smart technologies for mechanical engineering*. Vol. 25. 2013, p. 26.

[15] K.D. Kim. "Buckling behaviour of composite panels using the finite element method". In: *Composite Structures* 36.1 (1996), pp. 33–43. ISSN: 0263-8223. DOI: https://doi.org/10.1016/S0263-8223(96)00063-3. URL: https://www.sciencedirect.com/science/article/pii/S0263822396000633.

[16] Tahereh Korouzhdeh, Hamid Eskandari-Naddaf, and Morteza Gharouni-Nik. "An Improved Ant Colony Model for Cost Optimization of Composite Beams". In: *Applied Artificial Intelligence* 31.1 (2017). Cited by: 31, 44 – 63. DOI: 10.1080/08839514.2017.1296681. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85016396384&doi=10.1080%2f08839514.2017.1296681&partnerID=40&md5=65c418bde65f29d6ed2240cc8c565573.

[17] Christophe Lecoutre. "STR2: optimized simple tabular reduction for table constraints". In: *Constraints* 16 (2011), pp. 341–371.

[18] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. "Reasoning from last conflict (s) in constraint programming". In: *Artificial Intelligence* 173.18 (2009), pp. 1592–1614.

[19] Shuguang Li, Elena Sitnikova, Yuning Liang, and Abdul-Salam Kaddour. "The Tsai-Wu failure criterion rationalised in the context of UD composites". In: *Composites Part A: Applied Science and Manufacturing* 102 (2017), pp. 207–217.

[20] Michele Lombardi, Michela Milano, and Andrea Bartolini. "Empirical decision model learning". In: *Artificial Intelligence* 244 (2017), pp. 343–367.

[21] Vahid Ghaffari Mejlej, Paul Falkenberg, Eiko Türck, and Thomas Vietor. "Optimization of Variable Stiffness Composites in Automated Fiber Placement Process using Evolutionary Algorithms". In: *Procedia CIRP* 66 (2017). 1st CIRP Conference on Composite Materials Parts Manufacturing (CIRP CCMPM 2017), pp. 79–84. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2017.03.302. URL: https://www.sciencedirect.com/science/article/pii/S2212827117304900.

[22] L. Michel, P. Schaus, and P. Van Hentenryck. "MiniCP: a lightweight solver for constraint programming". In: *Mathematical Programming Computation* 13.1 (2021), pp. 133–184. DOI: 10.1007/s12532-020-00190-7. URL: https://doi.org/10.1007/s12532-020-00190-7.

[23] Antoons Miguel, Delecluse Augustin, Schaus Pierre, and Zein Samih. "Modeling and Solving a Composite Structure Design Problem with Constraint Programming". In: *CP2025* (2025).

[24] Nicholas Nethercote et al. "MiniZinc: Towards a Standard CP Modelling Language". In: *Principles and Practice of Constraint Programming – CP 2007*. Ed. by Christian Bessière. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543. ISBN: 978-3-540-74970-7.

[25] Hoang X. Nguyen, Jaehong Lee, Thuc P. Vo, and Domagoj Lanc. "Vibration and lateral buckling optimisation of thin-walled laminated composite channel-section beams". In: *Composite Structures* 143 (2016), pp. 84–92. ISSN: 0263-8223. DOI: https://doi.org/10.1016/j.compstruct.2016.02.011. URL: https://www.sciencedirect.com/science/article/pii/S0263822316300319.

[26] S. Nikbakt, S. Kamarian, and M. Shakeri. "A review on optimization of composite structures Part I: Laminated composites". In: *Composite Structures* 195 (2018), pp. 158–185. ISSN: 0263-8223. DOI: https://doi.org/10.1016/j.compstruct.2018.03.063. URL: https://www.sciencedirect.com/science/article/pii/S0263822317336279.

[27] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. "Propagation via lazy clause generation". In: *Constraints* 14.3 (2009), pp. 357–391. DOI: 10.1007/s10601-008-9064-x. URL: https://doi.org/10.1007/s10601-008-9064-x.

[28] Chung Hae Park, Woo Il Lee, Woo Suck Han, and Alain Vautrin. "Weight minimization of composite laminated plates with multiple constraints". In: *Composites Science and Technology* 63.7 (2003), pp. 1015–1026. ISSN: 0266-3538. DOI: https://doi.org/10.1016/S0266-3538(03)00014-9. URL: https://www.sciencedirect.com/science/article/pii/S0266353803000149.

[29] Guillaume Perez and Jean-Charles Régin. "Improving GAC-4 for table and MDD constraints". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2014, pp. 606–621.

[30] Laurent Perron, Frédéric Didier, and Steven Gay. "The CP-SAT-LP Solver". In: *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*. Ed. by Roland H. C. Yap. Vol. 280. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 3:1–3:2. ISBN: 978-3-95977-300-3. DOI: 10.4230/LIPIcs.CP.2023.3. URL: https://drops.dagstuhl.de/opus/volltexte/2023/19040.

[31] Laurent Perron and Frédéric Didier. *CP-SAT*. Version v9.11. Google, May 7, 2024. URL: https://developers.google.com/optimization/cp/cp_solver/.

[32] Laurent Perron and Vincent Furnon. *OR-Tools*. Version v9.11. Google, May 7, 2024. URL: https://developers.google.com/optimization/.

[33] Mutra Raja Sekhara Reddy, Bathini Sidda Reddy, Vanguru Nageswara Reddy, and Surisetty Sreenivasulu. "Prediction of natural frequency of laminated composite plates using artificial neural networks". In: *Engineering* 4.6 (2012), pp. 329–337.

[34] Florian Régin and Elisabetta De Maria. "Generative Constraint Programming Revisited". In: *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2024, pp. 18–26.

[35] Florencia Reguera and Víctor Hugo Cortínez. "Optimal design of composite thin-walled beams using simulated annealing". In: *Thin-Walled Structures* 104 (2016), pp. 71–81. ISSN: 0263-8231. DOI: https://doi.org/10.1016/j.tws.2016.03.001. URL: https://www.sciencedirect.com/science/article/pii/S026382311630057X.

[36] Jean-Charles Régin. "Generalized Arc Consistency for Global Cardinality Constraint." In: Jan. 1996, pp. 209–215.

[37] Pierre Schaus et al. *MaxiCP: A Constraint Programming Solver for Scheduling and Vehicle Routing*. 2024. URL: https://github.com/aia-uclouvain/maxicp.

[38] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. "Modeling and programming with gecode". In: *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael* 1 (2010).

[39] Paul Shaw. "Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems". In: *Principles and Practice of Constraint Programming — CP98*. Ed. by Michael Maher and Jean-Francois Puget. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 417–431. ISBN: 978-3-540-49481-2.

[40] Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. "An optimal arc consistency algorithm for a particular case of sequence constraint". In: *Constraints* 19 (2014), pp. 30–56.

[41] Juan Pablo Vielma. "Mixed integer linear programming formulation techniques". In: *Siam Review* 57.1 (2015), pp. 3–57.

[42] S. Zein, P. Basso, and S. Grihon. "A constraint satisfaction programming approach for computing manufacturable stacking sequences". In: *Computers & Structures* 136 (2014), pp. 56–63. ISSN: 0045-7949. DOI: https://doi.org/10.1016/j.compstruc.2014.01.016. URL: https://www.sciencedirect.com/science/article/pii/S0045794914000273.

[43] Samih Zein, Benoît Colson, and Stéphane Grihon. "A primal-dual backtracking optimization method for blended composite structures". In: *Structural and Multidisciplinary Optimization* 45.5 (2012), pp. 669–680. DOI: 10.1007/s00158-011-0716-x. URL: https://doi.org/10.1007/s00158-011-0716-x.

List of Figures

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**