Designing Languages to Optimise Industrial Processes

Tereso del Río[1]

SCML - RISC - July 6th, 2026

[1] Supported by the Austrian Research Promotion Agency (FFG), project no. 59218671: "InProSSA: Industrial Problem Solving Using Symbolic and Subsymbolic AI".

RISC Software GmbH

News Focus Areas Projects Research Career Company Contact EN | DE

InProSSA: Integration of symbolic and subsymbolic AI for industry

The exploratory project InProSSA investigates the question of whether different solution paradigms of symbolic and subsymbolic AI can be integrated into a generic, common modeling language in order to make the different concepts integratively usable. This allows the problem to be formulated independently of the solution approach, and the best solution method is automatically selected from a pool of existing methods.

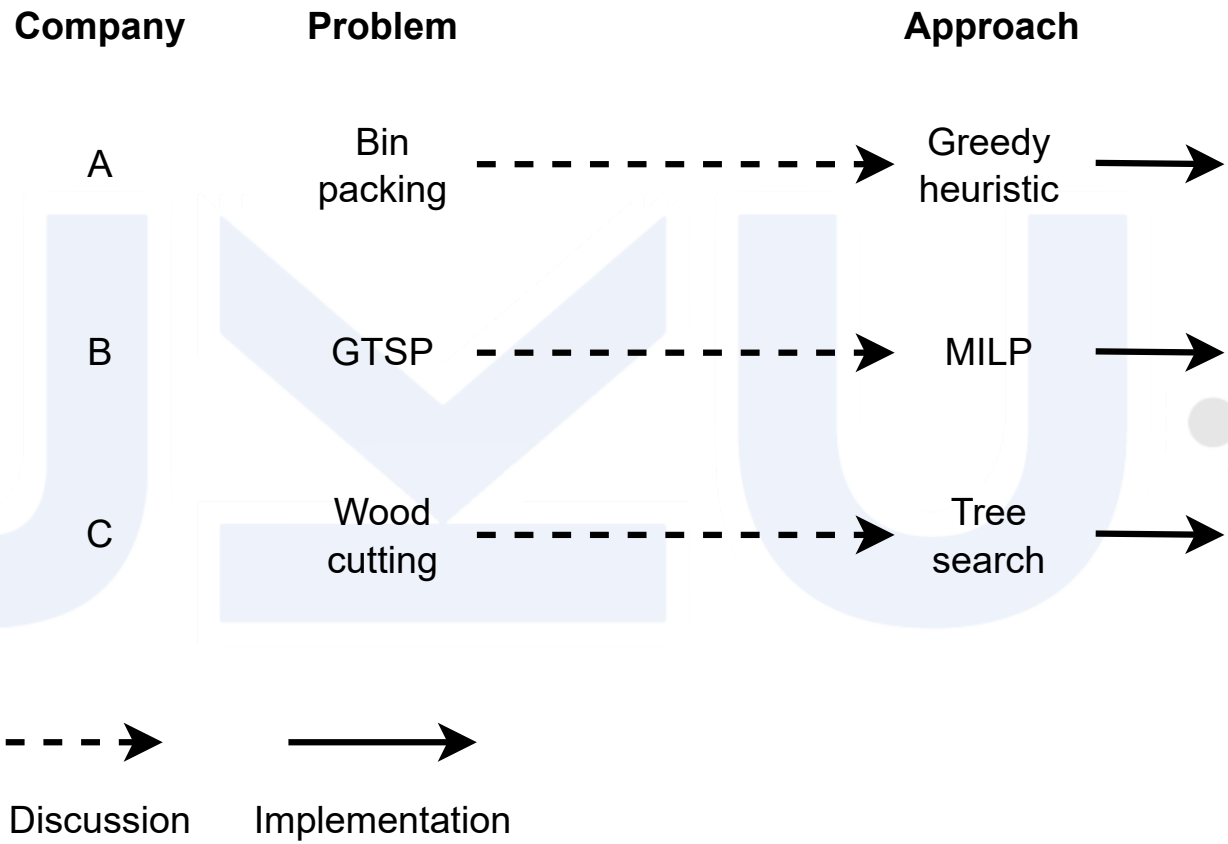
Project Partners

RISC Institut

JKU Institute for Symbolic Artificial Intelligence

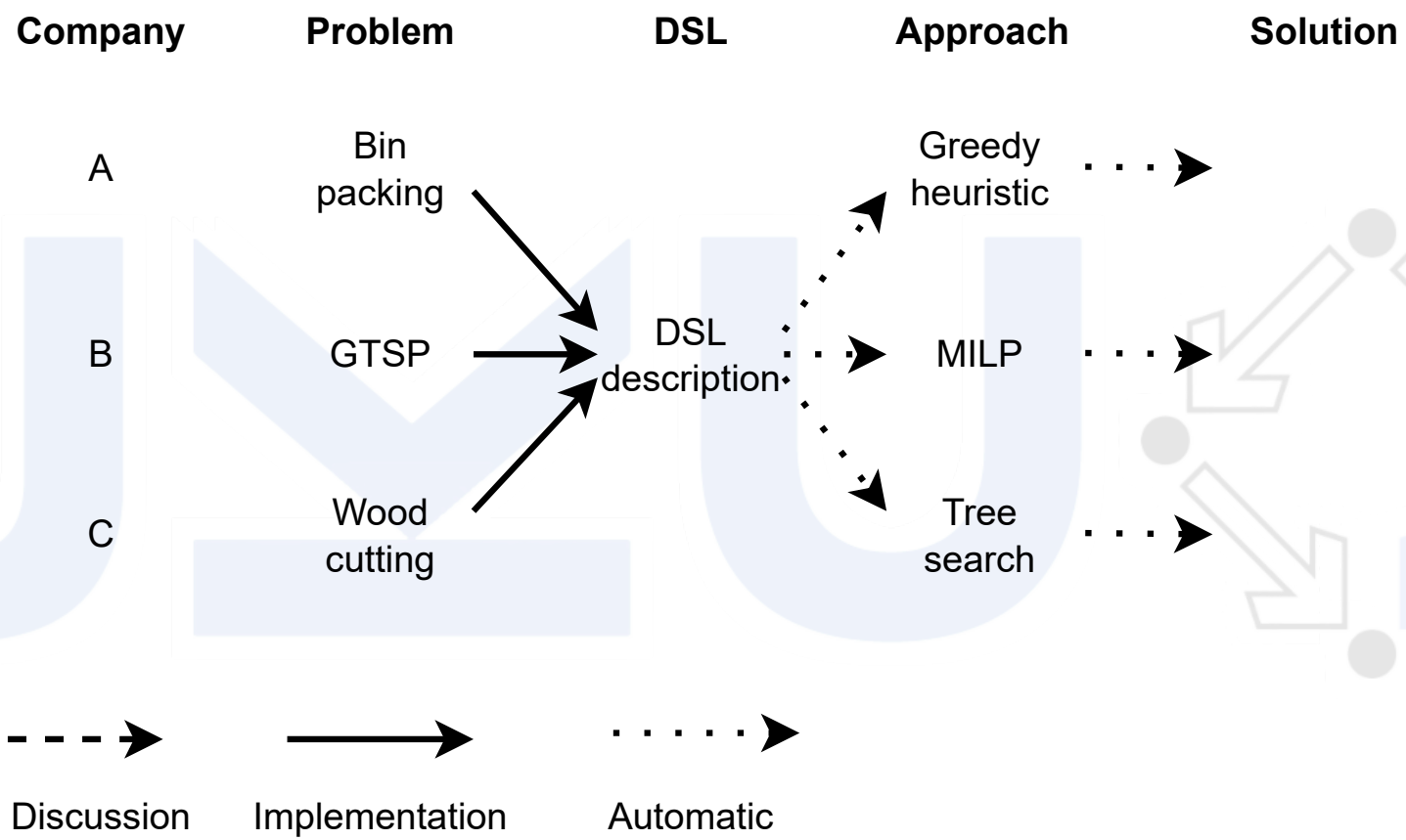
JKU JOHANNES KEPLER UNIVERSITÄT LINZ

CURRENT industrial partner pipeline



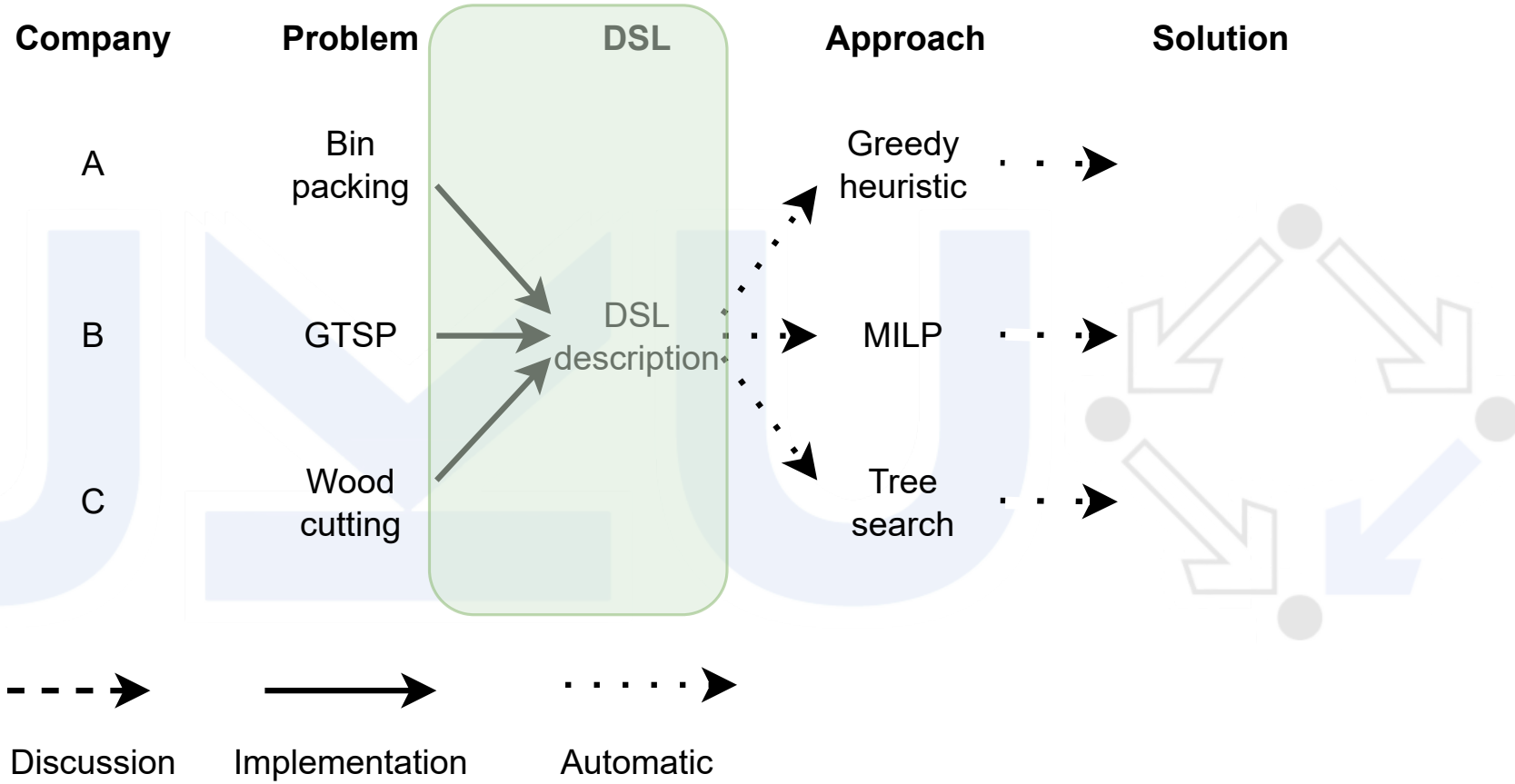
Figure

IDEAL industrial partner pipeline



Figure

Focus on DSL



Figure

MILP Language (MiniZinc)

MiniZinc example

```
int: N = 3;

array[1..N] of int: value = [4, 2, 7];

array[1..N] of int: weight = [3, 1, 4];

int: capacity = 5;


array[1..N] of var bool: take;


constraint sum(i in 1..N)(weight[i] * bool2int(take[i])) <= capacity;


solve maximize sum(i in 1..N)(value[i] * bool2int(take[i]));
```

Comparing Languages

MILP (MiniZinc)	**Ideal language**
Unfamiliar for developers.	Familiar programming style.
Abstracting constraints.	Describe behaviour naturally.
Modules are rigid.	Functions easily reusable.
Dubugging is very tricky.	Easily debuggable.
Nightmare to maintain.	Easily mantainable.
Tailored for solver.	Tailored for coder.

Comparing Languages

MILP (MiniZinc)	**Ideal language**
Unfamiliar for developers.	Familiar programming style.
Abstracting constraints.	Describe behaviour naturally.
Modules are rigid.	Functions easily reusable.
Dubugging is very tricky.	Easily debuggable.
Nightmare to maintain.	Easily mantainable.
Tailored for solver.	Tailored for coder.

Comparing Languages

MILP (MiniZinc)	**Ideal language**
Unfamiliar for developers.	Familiar programming style.
Abstracting constraints.	Describe behaviour naturally.
Modules are rigid.	Functions easily reusable.
Dubugging is very tricky.	Easily debuggable.
Nightmare to maintain.	Easily mantainable.
Tailored for solver.	Tailored for coder.

Comparing Languages

MILP (MiniZinc)	**Ideal language**
Unfamiliar for developers.	Familiar programming style.
Abstracting constraints.	Describe behaviour naturally.
Modules are rigid.	Functions easily reusable.
Dubugging is very tricky.	Easily debuggable.
Nightmare to maintain.	Easily mantainable.
Tailored for solver.	Tailored for coder.

Comparing Languages

MILP (MiniZinc)	**Ideal language**
Unfamiliar for developers.	Familiar programming style.
Abstracting constraints.	Describe behaviour naturally.
Modules are rigid.	Functions easily reusable.
Dubugging is very tricky.	Easily debuggable.
Nightmare to maintain.	Easily mantainable.
Tailored for solver.	Tailored for coder.

Comparing Languages

MILP (MiniZinc)	**Ideal language**
Unfamiliar for developers.	Familiar programming style.
Abstracting constraints.	Describe behaviour naturally.
Modules are rigid.	Functions easily reusable.
Dubugging is very tricky.	Easily debuggable.
Nightmare to maintain.	Easily mantainable.
Tailored for solver.	Tailored for coder.

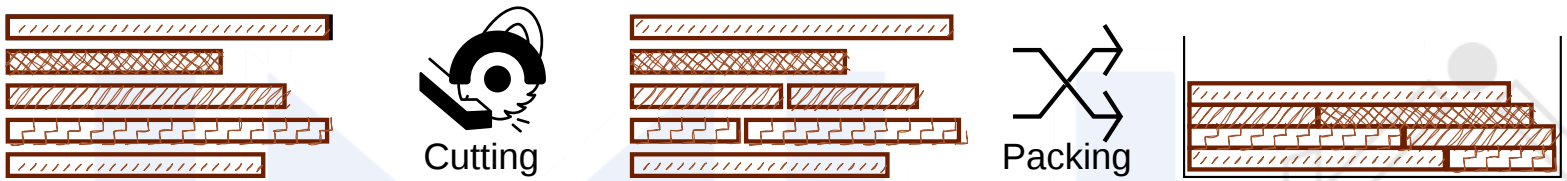
Comparing Languages

MILP (MiniZinc)	**Ideal language**
Unfamiliar for developers.	Familiar programming style.
Abstracting constraints.	Describe behaviour naturally.
Modules are rigid.	Functions easily reusable.
Dubugging is very tricky.	Easily debuggable.
Nightmare to maintain.	Easily mantainable.
Tailored for solver.	Tailored for coder.

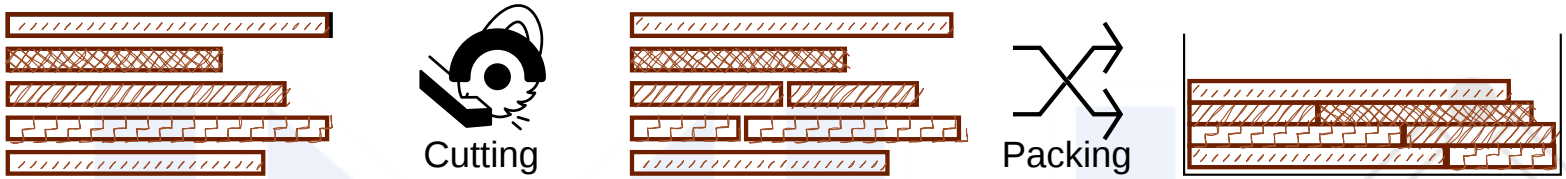
Comparing Languages

MILP (MiniZinc)	**Python checker-style**
Unfamiliar for developers.	Familiar programming style.
Abstracting constraints.	Describe behaviour naturally.
Modules are rigid.	Functions easily reusable.
Dubugging is very tricky.	Easily debuggable.
Nightmare to maintain.	Easily mantainable.
Tailored for solver.	Tailored for coder.

How to describe a problem



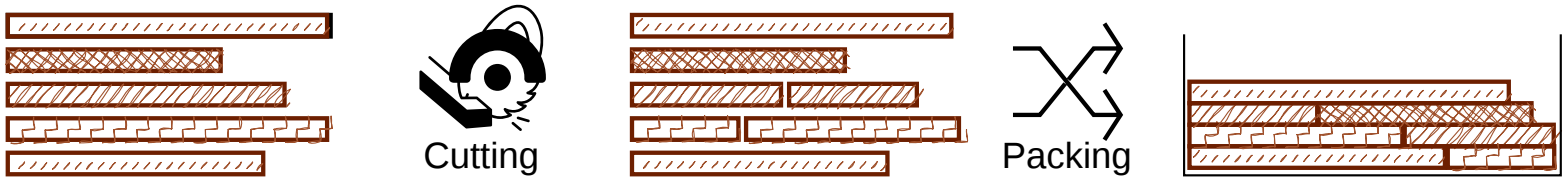
How to describe a problem



Python checker

given a solution,
computes **validity**
and **objective**

How to describe a problem



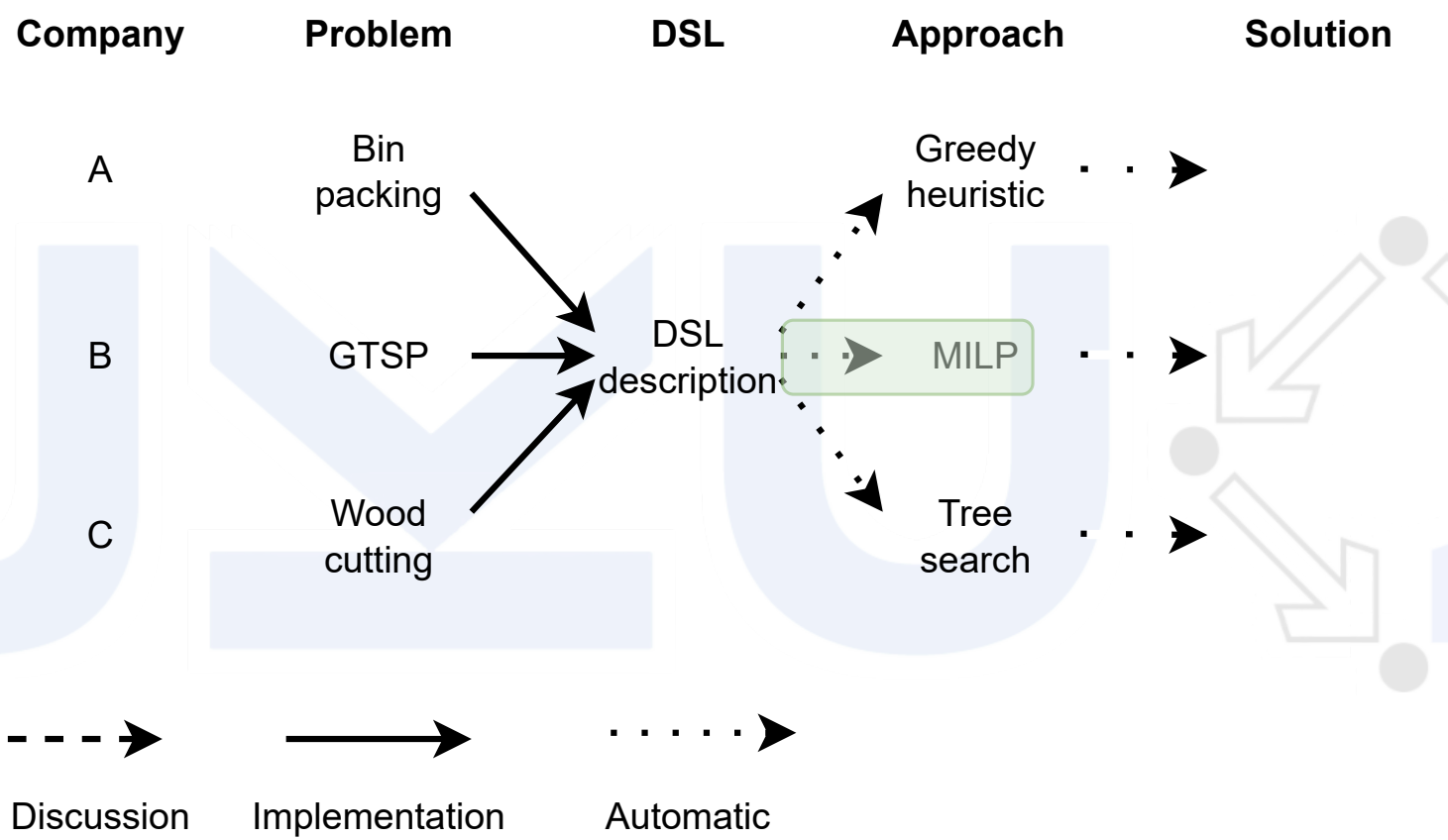
cutting and packing

```
LAYER_LENGTH = 17
ORIGINAL_BOARDS = [12, 15, 13, 10, 15]

CutPositions
cut_pieces, cutting_cost = cutting_stage(CutPositions)

Assignments
use_cost = packing_stage(Assignments, cut_pieces)

minimize(cutting_cost + use_cost)
```

Python checker

given a solution, computes **validity** and **objective**

Translating DSL to MILP



Figure

DSL ⇒ MiniZinc

Translations

```
a = 7          -> constraint a[1] = 7;
```

DSL ⇒ MiniZinc

Translations

```
a = 7          -> constraint a[1] = 7;

a = a - 1      -> constraint a[2] = a[1] - 1;
```

DSL ⇒ MiniZinc

Translations

```
a = 7           -> constraint a[1] = 7;

a = a - 1       -> constraint a[2] = a[1] - 1;

assert a > 5    -> constraint a[2] > 5;
```

DSL ⇒ MiniZinc

Translations

```
a = 7              -> constraint a[1] = 7;

a = a - 1          -> constraint a[2] = a[1] - 1;

assert a > 5       -> constraint a[2] > 5;

if a > 5:          -> constraint if (a[2] > 5) -> (a[3] = 3);
  a = 3               constraint if not(a[2] > 5) -> (a[3] = a[2]);
```

DSL ⇒ MiniZinc

Translations

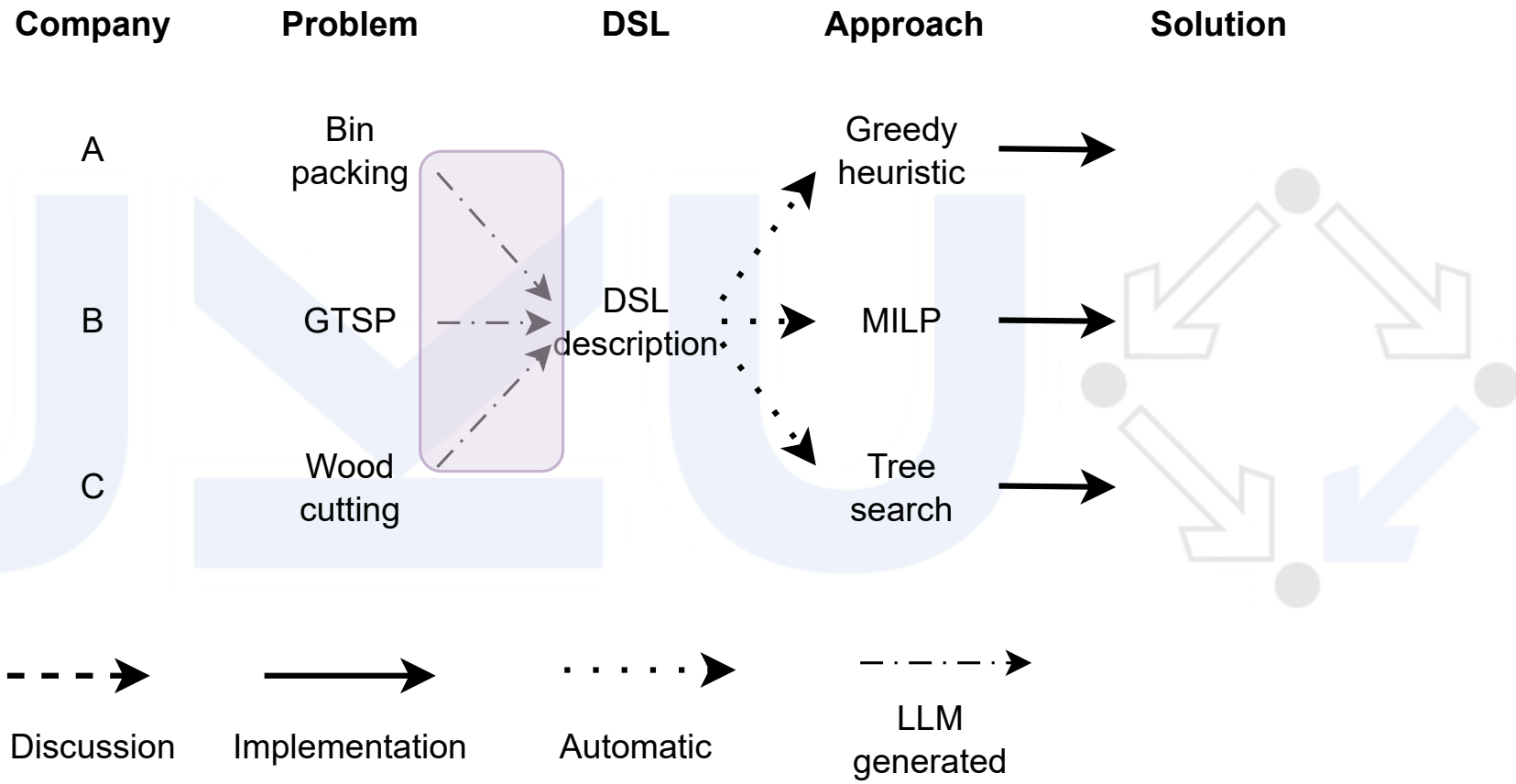
```
a = 7              -> constraint a[1] = 7;

a = a - 1          -> constraint a[2] = a[1] - 1;

assert a > 5       -> constraint a[2] > 5;

if a > 5:          -> constraint if (a[2] > 5) -> (a[3] = 3);
  a = 3               constraint if not(a[2] > 5) -> (a[3] = a[2]);

def add_one(a): -> predicate add_one(var int: a, var int: b) =
  b = a + 1           constraint b = a + 1;
  return b
```

DSL ⇒ MiniZinc

Translations

```
a = 7              -> constraint a[1] = 7;

a = a - 1          -> constraint a[2] = a[1] - 1;

assert a > 5       -> constraint a[2] > 5;

if a > 5:          -> constraint if (a[2] > 5) -> (a[3] = 3);
  a = 3               constraint if not(a[2] > 5) -> (a[3] = a[2]);

def add_one(a):    -> predicate add_one(var int: a, var int: b) =
  b = a + 1              constraint b = a + 1;
  return b

b = add_one(a)     -> constraint add_one(a, b)
```

Katharina's Master Thesis



Figure

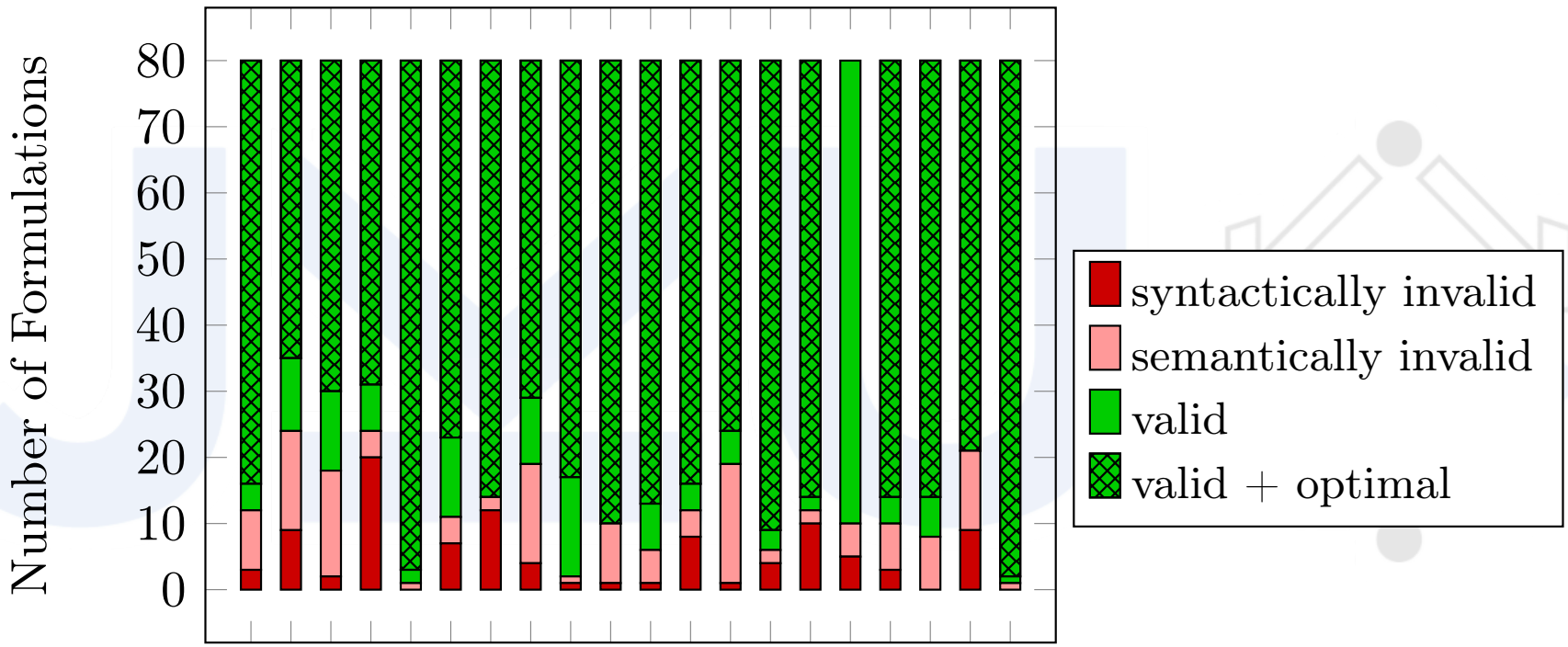
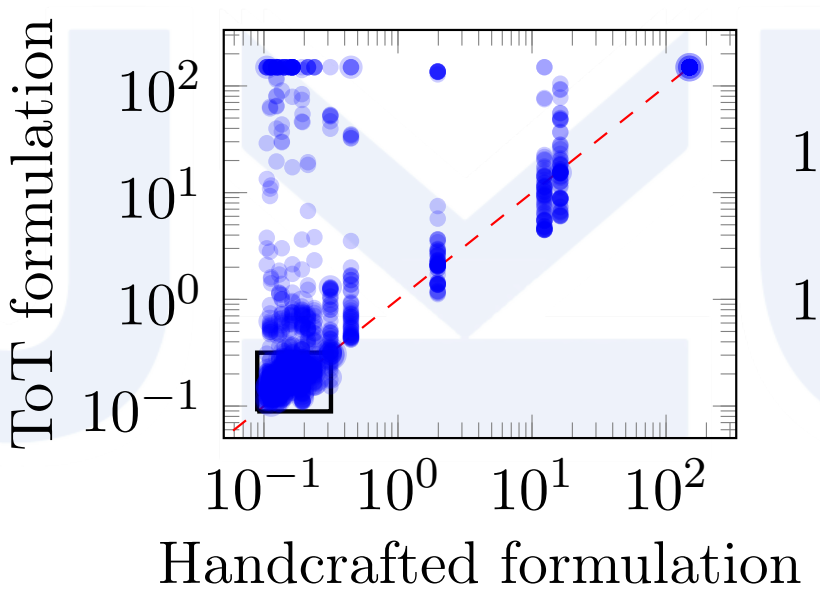
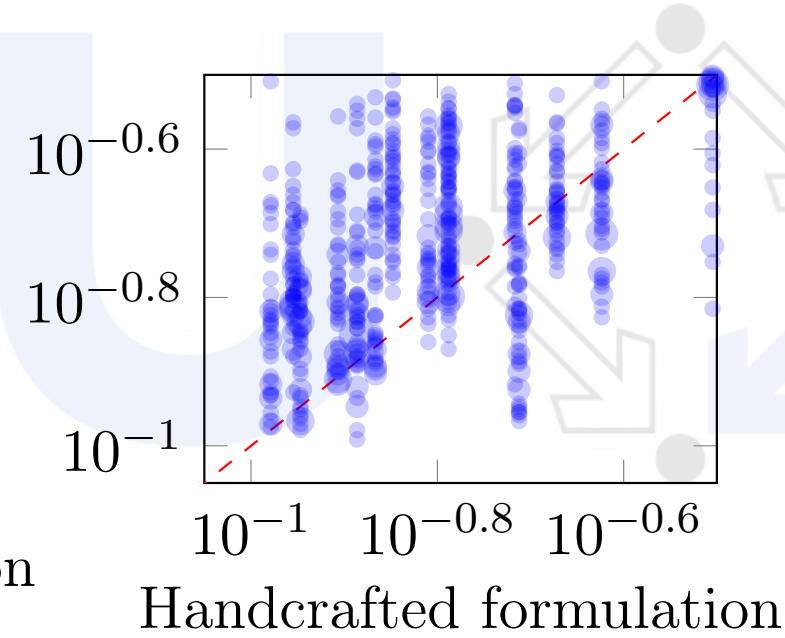


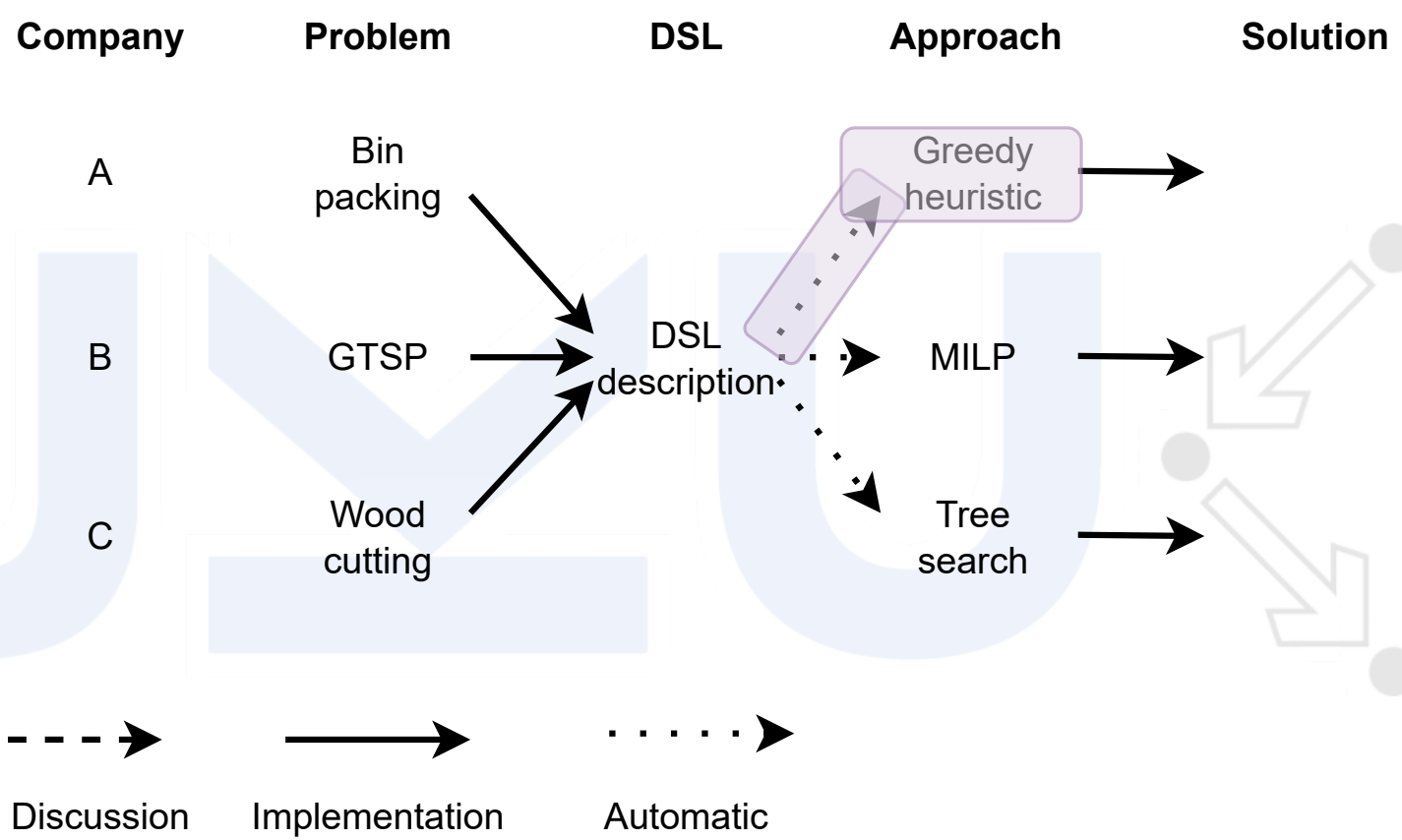
Figure: 80 formulations from 5 ToT per 2D-bin-packing instance.

Katharina's Master Thesis





Discovering heuristics



Figure

Thank You!

Happy to answer any questions

Escape Room

https://sites.google.com/view/risc-schnitzeljagd/