Taming Complexity of Industrial Printing Systems Using a Constraint-Based DSL

An Industrial Experience Report

LangDev'22 - September 28, 2022 - Aachen

Jasper Denkers - j.denkers@tudelft.nl,
Marvin Brunner, Louis van Gool, Jurgen Vinju, Andy Zaidman, Eelco Visser

Worldwide offer in books

delivered today!

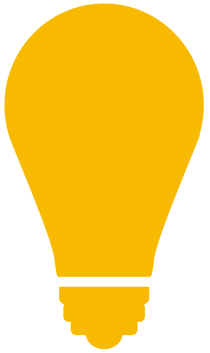
Digital Printing Systems

Control Software

- UI for operators to control printing systems

- Large configuration spaces
- For a product line of systems

Constraint Solving



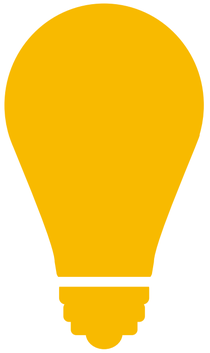
- SMT solving is a natural fit
- Writing SMT models by hand is cumbersome
-

Control Software

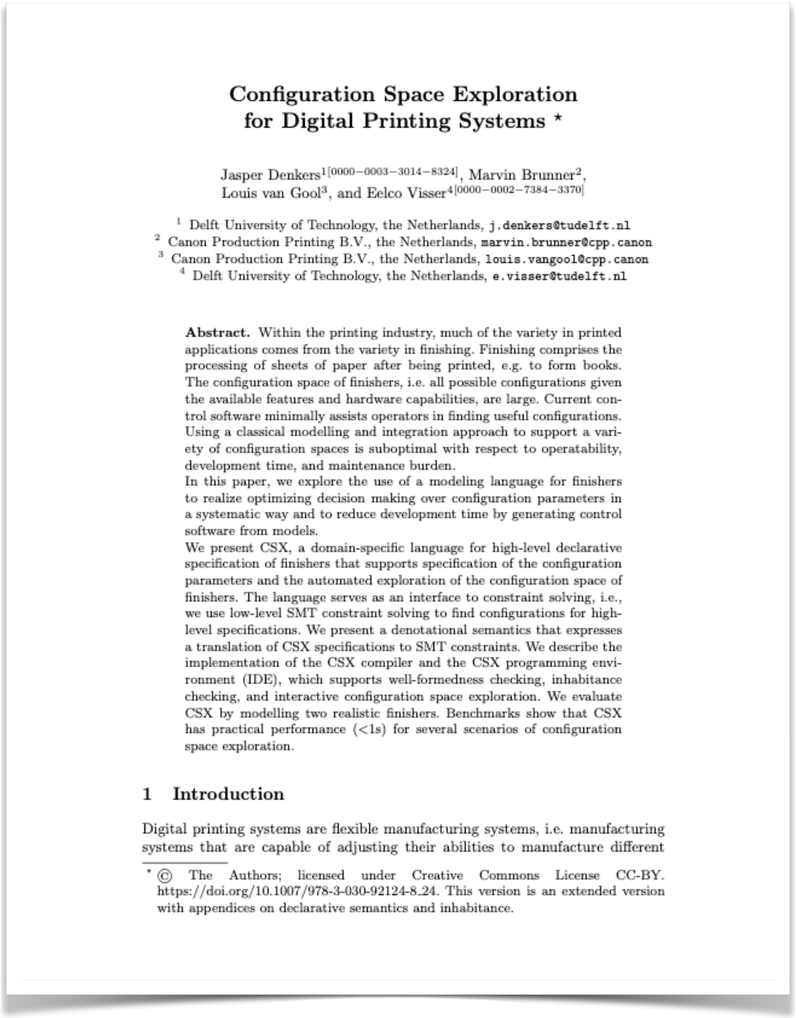
- UI for operators to control printing systems

- Large configuration spaces
- For a product line of systems

Language Engineering



- SMT solving is a natural fit
- Writing SMT models by hand is cumbersome
- DSL, with translation to SMT



Configuration Space Exploration for Digital Printing Systems *

Jasper Denkers[1][0000−0003−3014−8324], Marvin Brunner[2], Louis van Gool[3], and Eelco Visser[4][0000−0002−7384−3370]

[1] Delft University of Technology, the Netherlands, j.denkers@tudelft.nl
[2] Canon Production Printing B.V., the Netherlands, marvin.brunner@cpp.canon
[3] Canon Production Printing B.V., the Netherlands, louis.vangool@cpp.canon
[4] Delft University of Technology, the Netherlands, e.visser@tudelft.nl

Abstract. Within the printing industry, much of the variety in printed applications comes from the variety in finishing. Finishing comprises the processing of sheets of paper after being printed, e.g. to form books. The configuration space of finishers, i.e. all possible configurations given the available features and hardware capabilities, are large. Current control software minimally assists operators in finding useful configurations. Using a classical modelling and integration approach to support a variety of configuration spaces is suboptimal with respect to operatability, development time, and maintenance burden.

In this paper, we explore the use of a modeling language for finishers to realize optimizing decision making over configuration parameters in a systematic way and to reduce development time by generating control software from models.

We present CSX, a domain-specific language for high-level declarative specification of finishers that supports specification of the configuration parameters and the automated exploration of the configuration space of finishers. The language serves as an interface to constraint solving, i.e., we use low-level SMT constraint solving to find configurations for high-level specifications. We present a denotational semantics that expresses a translation of CSX specifications to SMT constraints. We describe the implementation of the CSX compiler and the CSX programming environment (IDE), which supports well-formedness checking, inhabitance checking, and interactive configuration space exploration. We evaluate CSX by modelling two realistic finishers. Benchmarks show that CSX has practical performance (<1s) for several scenarios of configuration space exploration.

1 Introduction

Digital printing systems are flexible manufacturing systems, i.e. manufacturing systems that are capable of adjusting their abilities to manufacture different

 https://doi.org/10.1007/978-3-030-92124-8_24. This version is an extended version with appendices on declarative semantics and inhabitance.

Introduction of CSX 1.0 (SEFM'22)

declarative semantics, denotational semantics in terms of SMT constraints

Taming Complexity of Industrial Printing Systems Using a Constraint-Based DSL — An Industrial Experience Report

Jasper Denkers
j.denkers@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Marvin Brunner
marvin.brunner@cpp.canon
Canon Production Printing
Venlo, The Netherlands

Louis van Gool
louis.vangool@cpp.canon
Canon Production Printing
Venlo, The Netherlands

Jurgen J. Vinju
jurgen.vinju@cwi.nl
CWI, Amsterdam/TU Eindhoven
The Netherlands

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Eelco Visser
e.visser@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Abstract

Flexible printing systems are highly complex systems that consist of printers, that print individual sheets of paper, and finishing equipment, that processes sheets after printing, e.g., assembling a book. Integrating finishing equipment with printers involves the development of control software that configures the devices, taking hardware constraints into account. This control software is highly complex to realize due to (1) the intertwined nature of printing and finishing, (2) the large variety of printing and finishing options, and (3) the variety of underlying hardware in the product line of finishers.

We have developed a domain-specific language called CSX that offers an interface to constraint solving specific to the printing domain. We use it to model printing devices and to automatically derive constraint solver-based environments for automatic configuration. We evaluate CSX on its coverage of the printing domain in an industrial context, and we report on lessons learned on using a constraint-based DSL in an industrial context.

Keywords: digital printing systems, domain-specific languages, constraint programming, industrial experiences

1 Introduction

What if we could have the worldwide offer in books, delivered tomorrow, at exceptionally low cost? Keeping all these books in stock is not an option because of storage prices. This is where *flexible printing systems* come in. With a flexible printing system, any book can be printed on demand and delivered to your home the same day. Such a printing system can be adjusted to print books with varying sizes and binding methods. To make this feasible for the operator of such a system, we need control software that supports configuring the printing system based on a description of the end product. This involves the process of *configuration space exploration*: finding a valid configuration that specifies the complete manufacturing process (the input materials, the device parameters, and the end product).

Developing control software with support for configuration space exploration is complex, because it needs to take many interdependent hardware details into account. This leads to handwritten software implementations that handle many individual cases non-systematically, while still not covering all possible configurations. The corresponding user interfaces of devices partially assist operators in finding configurations, but many aspects still require manual configuration. Moreover, such control software implementations are difficult to maintain, and this problem is further amplified by the product line of printing systems being large and diverse.

Canon Production Printing initiated a collaboration with Delft University of Technology to explore a model-driven approach for developing control software to tackle two challenges. First, realizing environments for configuration space exploration that is automatic and complete (i.e., covers all possible configurations). Second, cope with the variability in the product line of printing systems; besides devices that produce books, there are many others that, e.g., produce magazines, packaging, or decoration.

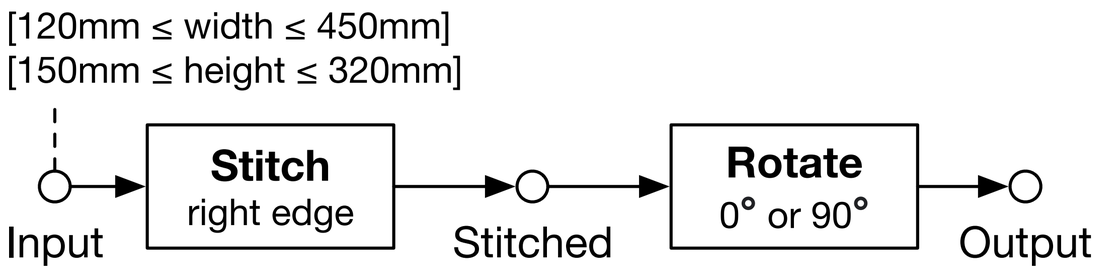
Constraint solving is a natural fit for developing control software with automatic configuration. By modelling printing systems as constraint models, we can use constraint solvers to achieve automatic configuration space exploration. A solution of the constraint model would correspond to a configuration for the printing system. Solvers can also find *optimal* solutions and thus optimal configurations, e.g., for objectives such as minimizing paper waste or maximizing print productivity. Therefore, we explore the usage of constraint modelling in realizing the next generation control software.

However, modelling a digital printing system — including all details of the mechanics — in a generic constraint modelling language is tedious, because it involves low-level modelling. Using a domain-specific language (DSL) for modelling configuration spaces has the potential of tackling this

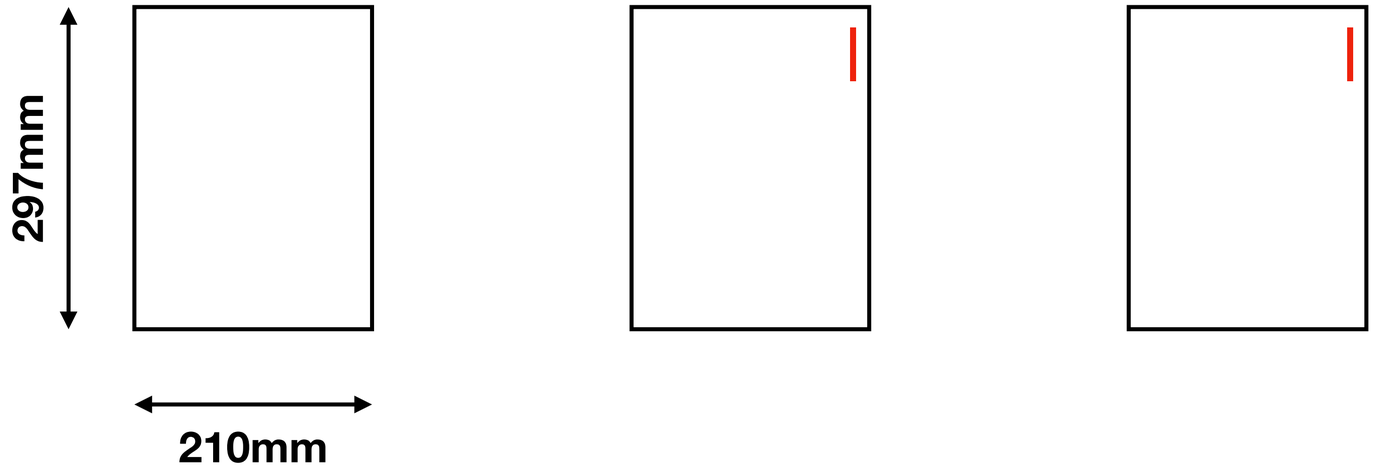
Evaluating CSX 2.0 in practice (under submission)

Extending CSX's coverage, evaluation of coverage, accuracy, performance, and relevance

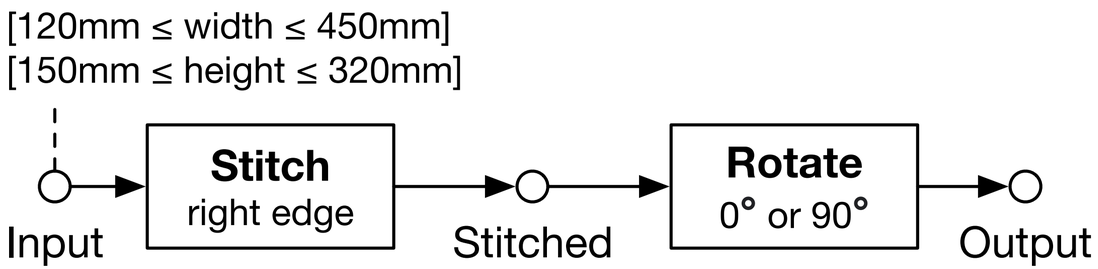
Configuration Space of Edge Stitching



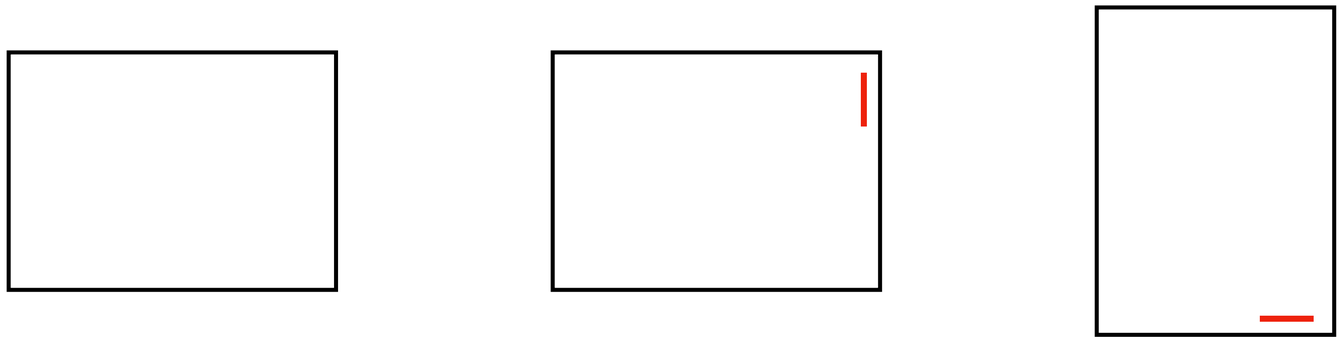
Scenario: input A4 portrait, rotate 0°



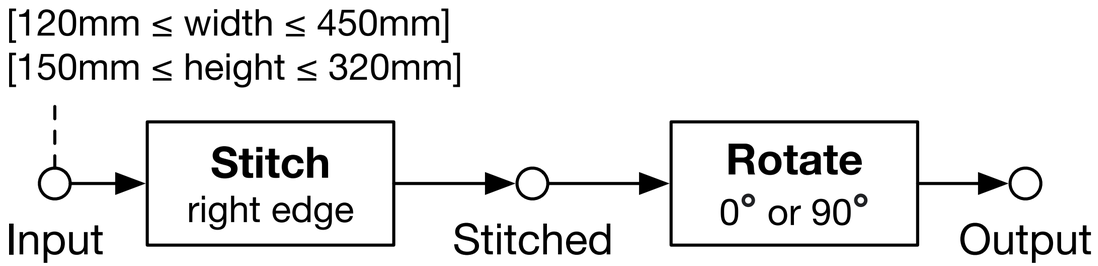
Configuration Space of Edge Stitching



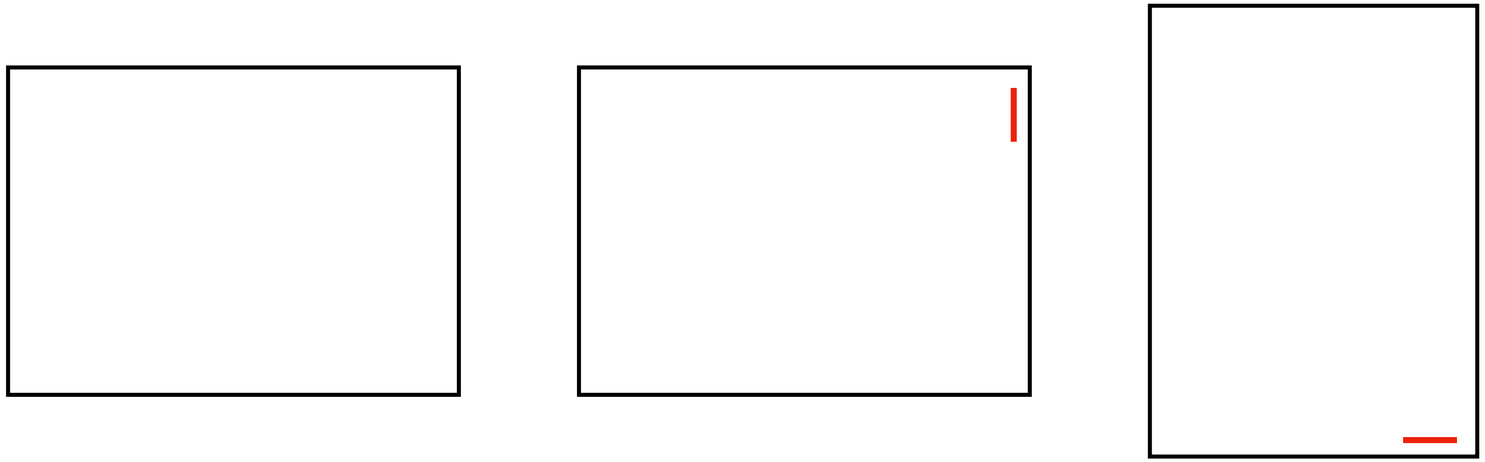
Scenario: output A4 portrait, edge at bottom



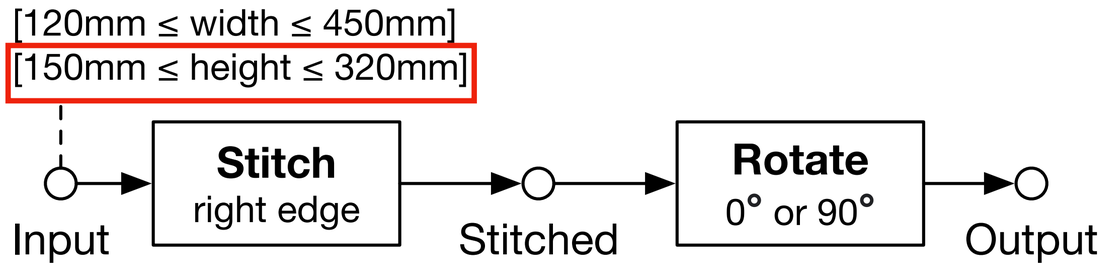
Configuration Space of Edge Stitching



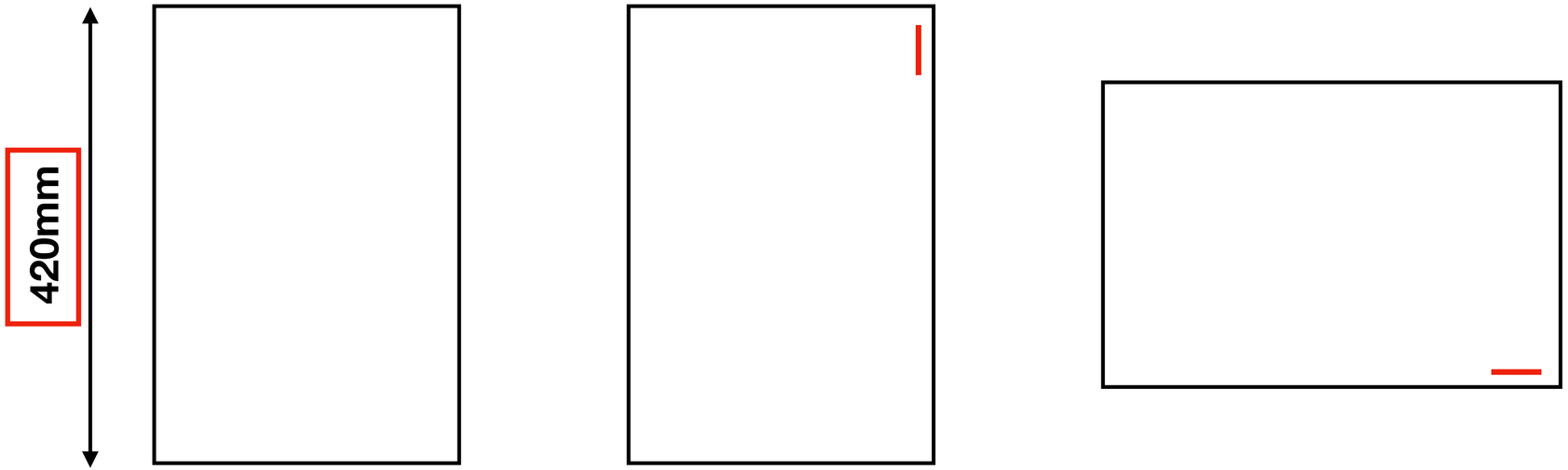
Scenario: output A3 portrait, edge at bottom



Configuration Space of Edge Stitching



Scenario: output A3 landscape, edge at bottom



Edge Stitching in CSX

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}

action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}

device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]

  component stitcher = Stitcher(input, stitched) {}

  // This device can only stitch on the right edge
  [stitcher.stitchEdge == right]

  location stitched: StitchedStack

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  [output.stitchEdge == orientate(stitched.stitchEdge, rotation)]

  location output: StitchedStack
}
```

Edge Stitching in CSX

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}

action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}

device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]

  component stitcher = Stitcher(input, stitched) {}

  // This device can only stitch on the right edge
  [stitcher.stitchEdge == right]

  location stitched: StitchedStack

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  [output.stitchEdge == orientate(stitched.stitchEdge, rotation)]

  location output: StitchedStack
}
```

User-defined types

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}
```

Edge Stitching in CSX

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}

action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}

device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]

  component stitcher = Stitcher(input, stitched) {}

  // This device can only stitch on the right edge
  [stitcher.stitchEdge == right]

  location stitched: StitchedStack

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  [output.stitchEdge == orientate(stitched.stitchEdge, rotation)]

  location output: StitchedStack
}
```

Actions

```
action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}
```

Edge Stitching in CSX

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}

action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}

device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]

  component stitcher = Stitcher(input, stitched) {}

  // This device can only stitch on the right edge
  [stitcher.stitchEdge == right]

  location stitched: StitchedStack

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  [output.stitchEdge == orientate(stitched.stitchEdge, rotation)]

  location output: StitchedStack
}
```

Devices

```
device EdgeStitcher {
  location input: list<Sheet>

  ...



  location stitched: StitchedStack

  ...

  parameter rotation: orientation

  ...

  location output: StitchedStack
}
```

Edge Stitching in CSX

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}

action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}

device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]

  component stitcher = Stitcher(input, stitched) {}

  // This device can only stitch on the right edge
  [stitcher.stitchEdge == right]

  location stitched: StitchedStack

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  [output.stitchEdge == orientate(stitched.stitchEdge, rotation)]

  location output: StitchedStack
}
```

Devices

```
device EdgeStitcher {
  location input: list<Sheet>

  ...

  component stitcher = Stitcher(input, stitched) {}

  location stitched: StitchedStack

  ...

  parameter rotation: orientation

  ...

  location output: StitchedStack
}
```

Edge Stitching in CSX

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}

action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}

device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]

  component stitcher = Stitcher(input, stitched) {}

  // This device can only stitch on the right edge
  [stitcher.stitchEdge == right]

  location stitched: StitchedStack

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  [output.stitchEdge == orientate(stitched.stitchEdge, rotation)]

  location output: StitchedStack
}
```

Devices

```
device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]



  ...


}
```

Edge Stitching in CSX

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}

action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}

device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]

  component stitcher = Stitcher(input, stitched) {}

  // This device can only stitch on the right edge
  [stitcher.stitchEdge == right]

  location stitched: StitchedStack

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  [output.stitchEdge == orientate(stitched.stitchEdge, rotation)]

  location output: StitchedStack
}
```

Devices

```
device EdgeStitcher {
  ...

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  ...

}
```

Edge Stitching in CSX

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}

action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}

device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]

  component stitcher = Stitcher(input, stitched) {}

  // This device can only stitch on the right edge
  [stitcher.stitchEdge == right]

  location stitched: StitchedStack

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  [output.stitchEdge == orientate(stitched.stitchEdge, rotation)]

  location output: StitchedStack
}
```

Testing

```
scenario device EdgeStitcher
         config size(output.sheets) == 2 {
         config output.sheets[1].width  == 2100
         config output.sheets[1].height == 2970
         config output.sheets[2].width  == 2100
         config output.sheets[2].height == 2970 {

  // Stitch on bottom edge requires rotation of 90 degrees
  test config output.stitchEdge == bottom {
    [rotation == rot90]
  }

}
```

Edge Stitching in CSX

```
type Sheet {
  width: int,  [width > 0],
  height: int, [height > 0]
}

type StitchedStack {
  sheets: list<Sheet>,
  stitchEdge: edge
}

action Stitcher(input: list<Sheet>,
                output: StitchedStack) {
  [size(input) >= 2] // At least two sheets required for stitching
  parameter stitchEdge: edge
  [output.sheets == input]
  [output.stitchEdge == stitchEdge]
}

device EdgeStitcher {
  location input: list<Sheet>

  [size(input) <= 10] // Max number of sheets

  [input.forall { sheet =>
    // Min and max sheet sizes
    sheet.width >= 1200 and sheet.height >= 1500 and
    sheet.width <= 4500 and sheet.height <= 3200
  }]

  component stitcher = Stitcher(input, stitched) {}

  // This device can only stitch on the right edge
  [stitcher.stitchEdge == right]

  location stitched: StitchedStack

  parameter rotation: orientation
  [rotation == rot0 or rotation == rot90]

  [output.sheets.forall { sheet =>
    (rotation == rot0 implies sheet == stitched.sheets[index])
    and
    (rotation == rot90 implies (
      // Swap width and height in case of 90 degrees rotation
      sheet.width  == stitched.sheets[index].height and
      sheet.height == stitched.sheets[index].width
    ))
  }]
  [size(output.sheets) == size(stitched.sheets)]

  [output.stitchEdge == orientate(stitched.stitchEdge, rotation)]

  location output: StitchedStack
}
```

Testing

```
scenario device EdgeStitcher
         config size(output.sheets) == 2 {

  // A3 sheets in portrait orientation with stitch on
  // right edge is not possible
  test config output.sheets[1].width  == 2970
       config output.sheets[1].height == 4200
       config output.sheets[2].width  == 2970
       config output.sheets[2].height == 4200
       config output.stitchEdge == right {
    fails
  }

}
```

Translation to SMT constraints

```
array [1..2] of var  −1..1 : Top  = [0,1];
array [1..2] of var  −1..1 : Right  = [1,0];
array [1..2] of var  −1..1 : Bottom  = [0,−1];
array [1..2] of var  −1..1 : Left  = [−1,0];
predicate isEdge(array [1..2] of var  −1..1 : e)  = e = Top \/ e = Right \/ e = Bottom \/ e = Left;
array [1..2,1..2] of −1..1 : Rot0  = [|1,0|0,1|];
array [1..2,1..2] of −1..1 : Rot90  = [|0,1|−1,0|];
array [1..2,1..2] of −1..1 : Rot180  = [|−1,0|0,−1|];
array [1..2,1..2] of −1..1 : Rot270  = [|0,−1|1,0|];
array [1..2,1..2] of −1..1 : Flip0  = [|1,0|0,−1|];
array [1..2,1..2] of −1..1 : Flip90  = [|0,−1|−1,0|];
array [1..2,1..2] of −1..1 : Flip180  = [|−1,0|0,1|];
array [1..2,1..2] of −1..1 : Flip270  = [|0,1|1,0|];
predicate isOrientation(array [1..2,1..2] of var  −1..1 : o)  = o = Rot0 \/ o = Rot90 \/ o = Rot180 \/ o = Rot270 \/ o = Flip180 \/ o = Flip0 \/ o = Flip90 \/ o = Flip270;
function array [1..2] of var  −1..1: orientateEdge(array [1..2] of var  −1..1 : e,array [1..2,1..2] of var  −1..1 : o)  = array1d(1 .. 2,[e[1] * o[1,1] + e[2] * o[1,2],e[1] * o[2,1]
var  0..10 : input_size;
array [1..10] of var  int : input_width;
constraint forall(i in 1 .. 10) (i > input_size -> input_width[i] = 0);
array [1..10] of var  int : input_height;
constraint forall(i in 1 .. 10) (i > input_size -> input_height[i] = 0);
constraint forall(i in 1 .. 10) (i <= input_size -> input_width[i] > 0);
constraint forall(i in 1 .. 10) (i <= input_size -> input_height[i] > 0);
constraint input_size <= 10;
constraint forall(sheet in 1 .. 10) (sheet <= input_size -> input_width[sheet] >= 1200 /\ input_height[sheet] >= 1500 /\ input_width[sheet] <= 4500 /\ input_height[sheet] <= 3200);
constraint input_size >= 2;
array [1..2] of var  −1..1 : stitcher_stitchEdge;
constraint isEdge(stitcher_stitchEdge);
constraint stitched_sheets_size == input_size /\ (stitched_sheets_width == input_width /\ stitched_sheets_height == input_height);
constraint stitched_stitchEdge == stitcher_stitchEdge;
constraint stitcher_stitchEdge == Right;
var  0..10 : stitched_sheets_size;
array [1..10] of var  int : stitched_sheets_width;
constraint forall(i in 1 .. 10) (i > stitched_sheets_size -> stitched_sheets_width[i] = 0);
array [1..10] of var  int : stitched_sheets_height;
constraint forall(i in 1 .. 10) (i > stitched_sheets_size -> stitched_sheets_height[i] = 0);
array [1..2] of var  −1..1 : stitched_stitchEdge;
constraint forall(i in 1 .. 10) (i <= stitched_sheets_size -> stitched_sheets_width[i] > 0);
constraint forall(i in 1 .. 10) (i <= stitched_sheets_size -> stitched_sheets_height[i] > 0);
constraint isEdge(stitched_stitchEdge);
array [1..2,1..2] of var  −1..1 : rotation;
constraint isOrientation(rotation);
constraint rotation == Rot0 \/ rotation == Rot90;
constraint forall(sheet in 1 .. 10) (sheet <= output_sheets_size -> (rotation == Rot0 -> output_sheets_width[sheet] == stitched_sheets_width[sheet] /\ output_sheets_height[sheet] ==
constraint output_sheets_size == stitched_sheets_size;
constraint output_stitchEdge == orientateEdge(stitched_stitchEdge,rotation);
var  0..10 : output_sheets_size;
array [1..10] of var  int : output_sheets_width;
constraint forall(i in 1 .. 10) (i > output_sheets_size -> output_sheets_width[i] = 0);
array [1..10] of var  int : output_sheets_height;
constraint forall(i in 1 .. 10) (i > output_sheets_size -> output_sheets_height[i] = 0);
array [1..2] of var  −1..1 : output_stitchEdge;
constraint forall(i in 1 .. 10) (i <= output_sheets_size -> output_sheets_width[i] > 0);
constraint forall(i in 1 .. 10) (i <= output_sheets_size -> output_sheets_height[i] > 0);
constraint isEdge(output_stitchEdge)
```

expressed in **MiniZinc**:
a high-level, solver-independent
constraint modelling language

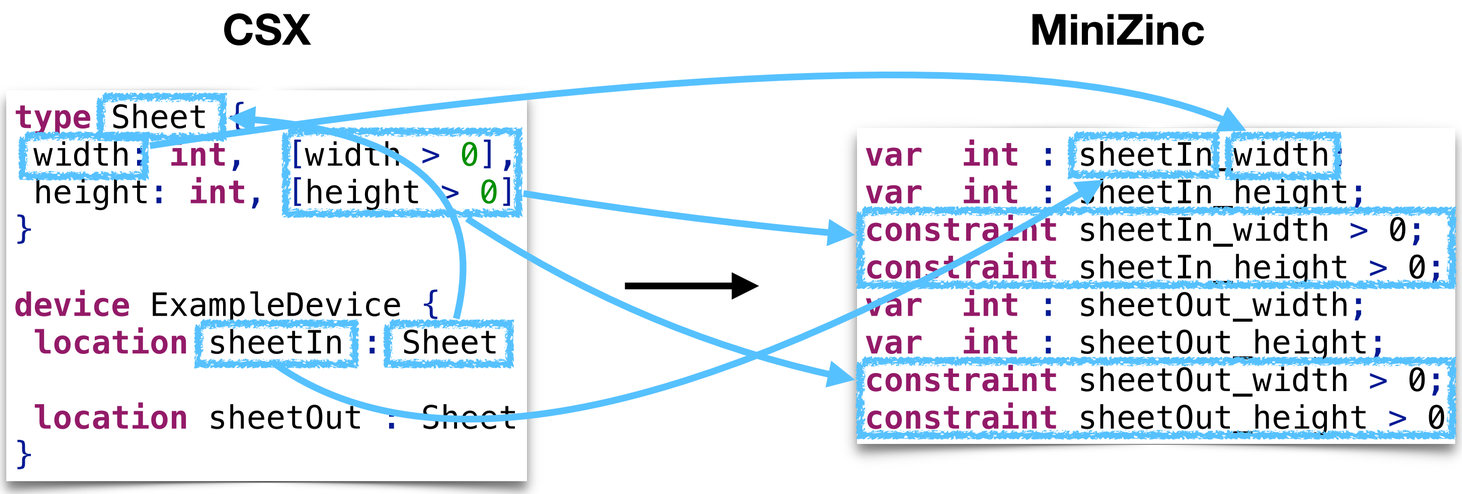
https://www.minizinc.org

OR Tools

Gecode

Chuffed

From CSX to SMT constraints



From Constraints to Configurations

CSX

```
type Sheet {
 width: int,  [width > 0],
 height: int, [height > 0]
}

device ExampleDevice {
 location sheetIn : Sheet

 location sheetOut : Sheet
}
```

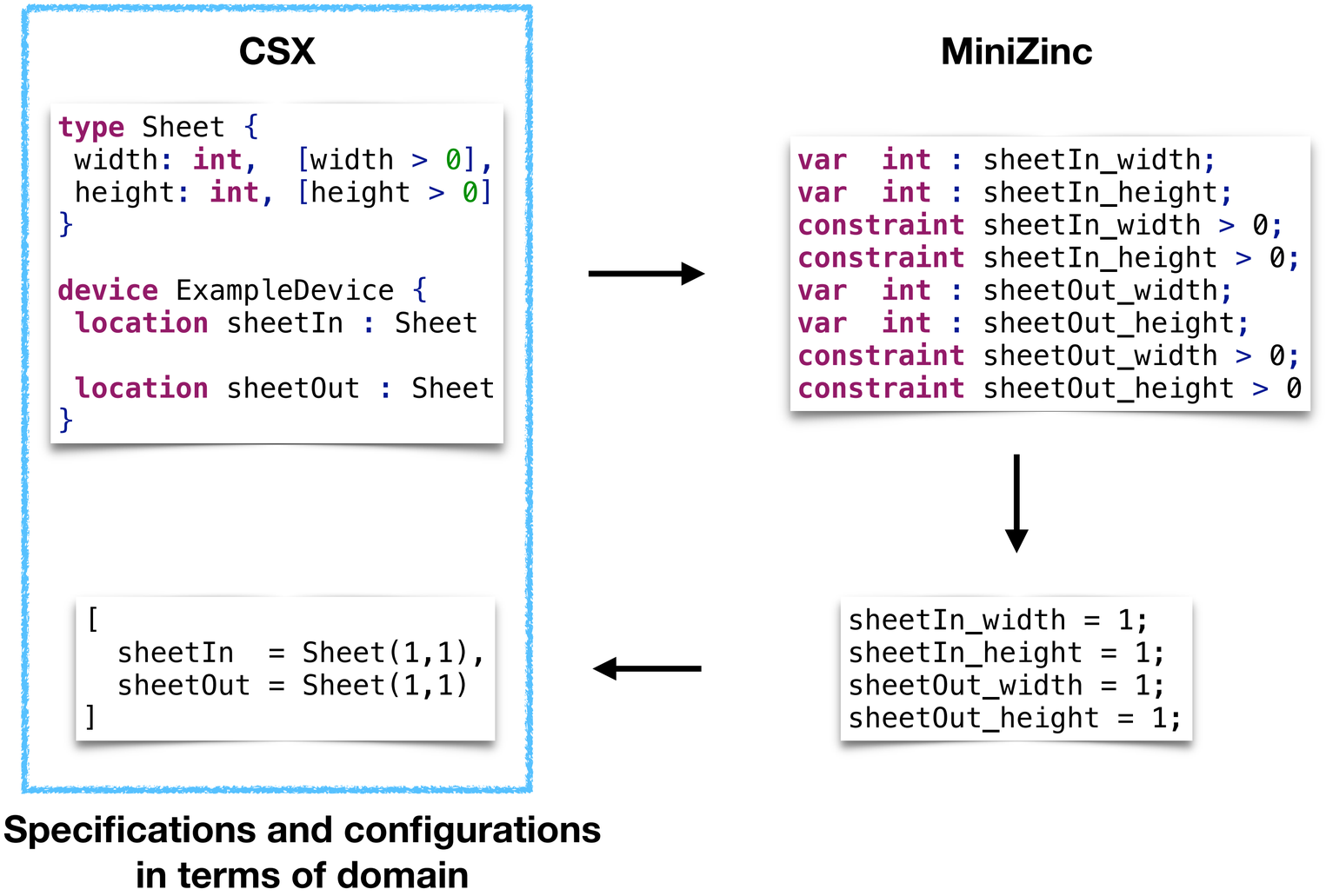
MiniZinc

```
var  int : sheetIn_width;
var  int : sheetIn_height;
constraint sheetIn_width > 0;
constraint sheetIn_height > 0;
var  int : sheetOut_width;
var  int : sheetOut_height;
constraint sheetOut_width > 0;
constraint sheetOut_height > 0
```

```
sheetIn_width = 1;
sheetIn_height = 1;
sheetOut_width = 1;
sheetOut_height = 1;
```

```
[
  sheetIn  = Sheet(1,1),
  sheetOut = Sheet(1,1)
]
```

Why Using a DSL?



Specifications and configurations in terms of domain

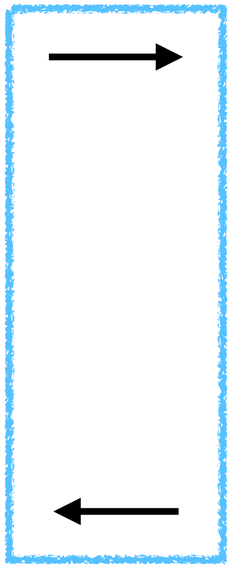
Why Using a DSL?

CSX

```
type Sheet {
 width: int,  [width > 0],
 height: int, [height > 0]
}

device ExampleDevice {
 location sheetIn : Sheet

 location sheetOut : Sheet
}
```

MiniZinc

```
var  int : sheetIn_width;
var  int : sheetIn_height;
constraint sheetIn_width > 0;
constraint sheetIn_height > 0;
var  int : sheetOut_width;
var  int : sheetOut_height;
constraint sheetOut_width > 0;
constraint sheetOut_height > 0
```



```
[
  sheetIn  = Sheet(1,1),
  sheetOut = Sheet(1,1)
]
```

```
sheetIn_width = 1;
sheetIn_height = 1;
sheetOut_width = 1;
sheetOut_height = 1;
```

Automatic mapping
to and from constraints

Why Using a DSL?

CSX

```
type Sheet {
 width: int,  [width > 0],
 height: int, [height > 0]
}

device ExampleDevice {
 location sheetIn : Sheet

 location sheetOut : Sheet
}
```

MiniZinc

```
var  int : sheetIn_width;
var  int : sheetIn_height;
constraint sheetIn_width > 0;
constraint sheetIn_height > 0;
var  int : sheetOut_width;
var  int : sheetOut_height;
constraint sheetOut_width > 0;
constraint sheetOut_height > 0
```

```
sheetIn_width = 1;
sheetIn_height = 1;
sheetOut_width = 1;
sheetOut_height = 1;
```

```
[
  sheetIn  = Sheet(1,1),
  sheetOut = Sheet(1,1)
]
```

Leverage the power of existing solvers

From Prototype to Application in Practice

- Early observations in applying CSX on realistic devices revealed domain coverage gaps:
  - Lists
  - Geometrical constructs
  - Functional style operators

Lists support

- Non-uniform stacks of sheets
- Upper bounds

```
type Sheet { width: int, height: int }
type Stack { sheets: list<Sheet> }
device D {
  location a: Stack
}
```

CSX spec

```
var  0..10 : a_sheets_size;
array [1..10] of var  int : a_sheets_width;
constraint forall(i in 1 .. 10) (i > a_sheets_size -> a_sheets_width[i] = 0);
array [1..10] of var  int : a_sheets_height;
constraint forall(i in 1 .. 10) (i > a_sheets_size -> a_sheets_height[i] = 0)
```

MiniZinc model

```
a = Stack([
  Sheet(2100, 2970), Sheet(2100, 2970)
])
```

CSX configuration

```
a_sheets_size = 2
a_sheets_width = [2100, 2100, 0, 0, 0, 0, 0, 0, 0, 0]
a_sheets_height = [2970, 2970, 0, 0, 0, 0, 0, 0, 0, 0]
```

MiniZinc solution

Geometrical constructs

- Edges, orientations, transformations
- Could be modelled by manually modelling linear algebra, but is cumbersome

```
device D {
  var a: edge
  var p: orientation

  [b == orientate(a,p)]

  var b: edge
}
```

Functional-style operators

- Bridge the gap between declarative/constraint-based programming and a more familiar programming style
- Requires the introduction of variables to represent intermediate results

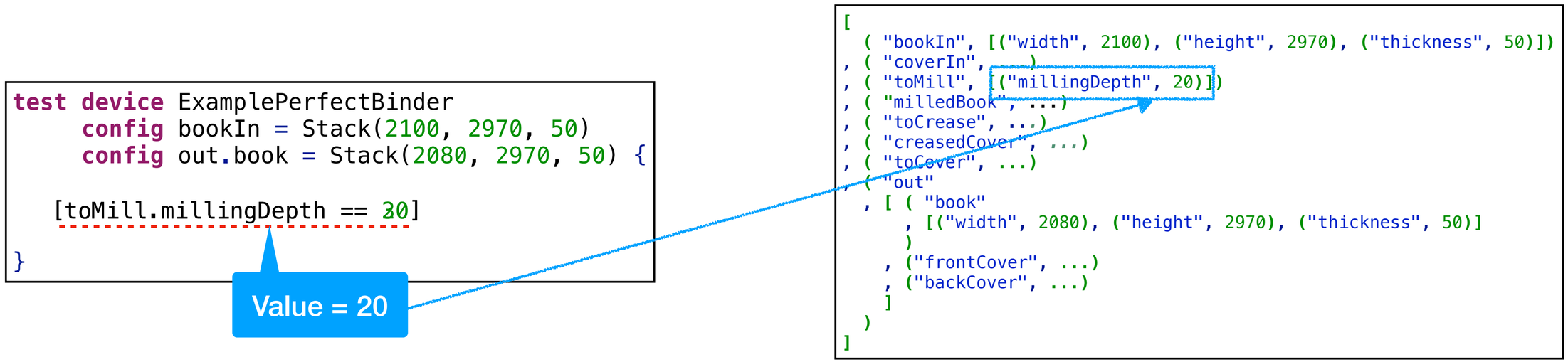
```
var bs: list<bool>

[reverse(bs)[1]]
```

```
var bs: list<bool>
var i1: list<bool>

[reverse(bs) == i1]
[i1[1]]
```

```
var bs: list<bool>
var i1: list<bool>

[bs.size == i1.size]
[bs.forall { x => i1[bs.size + 1 - index] = x }]
[i1[1]]
```

Implementation

- IDE with syntax highlighting, type checker, inhabitance checks
- Implemented using the Spoofax language workbench
- Interactive validation



Evaluating CSX 2.0 at Canon Production Printing

- Coverage => think aloud co-design sessions
- Performance => benchmarking

In the paper:

- Accuracy => testing for completeness and correctness
- Relevance => evaluate sufficiency and necessity

Evaluating Coverage

- Think aloud co-design sessions
- Together with a domain export, model cases of increasing complexity
- 7 iterations of modelling, gradually adding aspects

Iteration	Aspects introduced
1	*Uniform* stacks of sheets, device, physical limitations, validation
2	*Non-uniform* stacks of sheets
3	Input trays (incorrect)
4	Sheets must have equal height
5	Edge stichting, orientations
6	Input trays (correct)
7	Integrate input trays with edge stitching

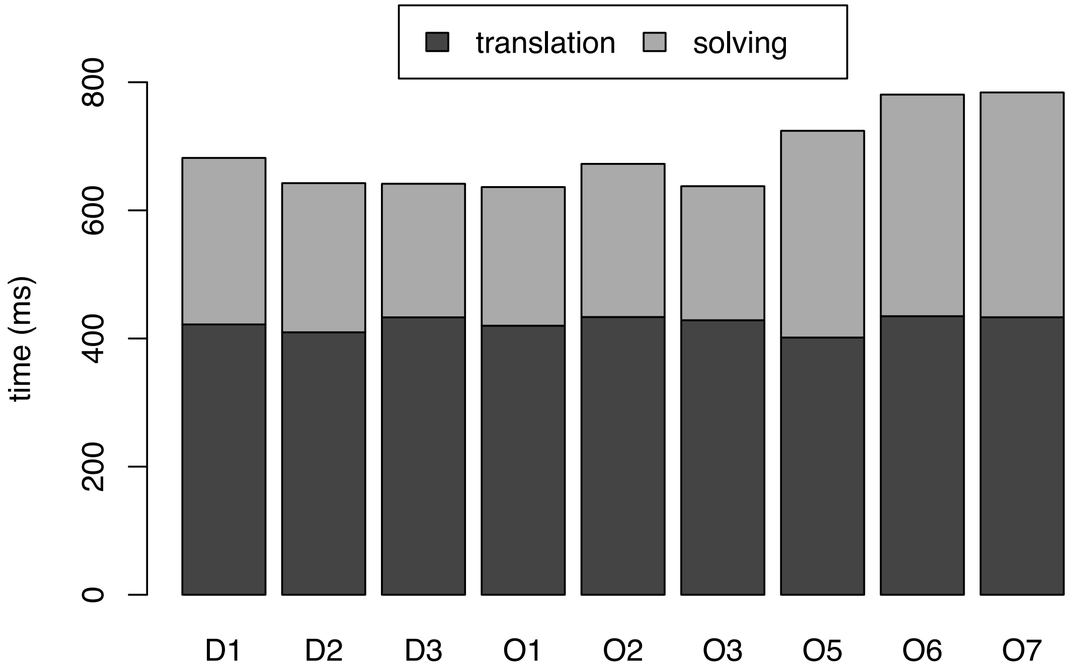
Evaluating Coverage

- (+) User-defined types enable modeling on a level of abstraction that corresponds to domain objects, while remaining flexible
- (+) Reporting configurations in terms of domain solves the problem of manually interpreting solutions
- (+) The lists construct contributes to covering non-uniform stacks of sheets of variability of e.g. the number of stitches.

Evaluating Coverage

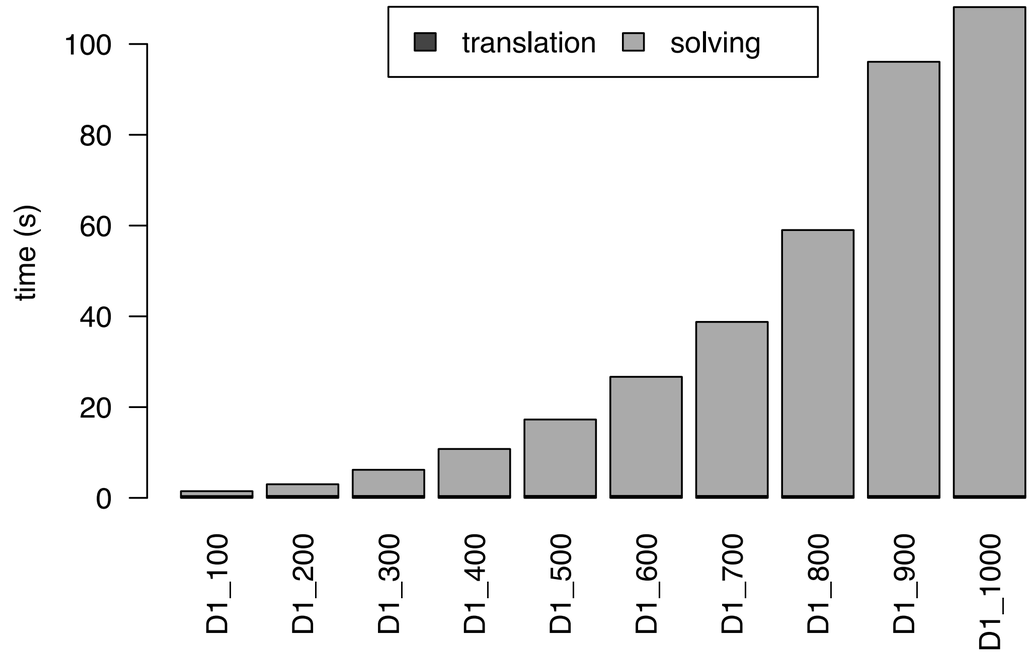
- (-) Units and precision are not covered, and configurations that are found need to be manually interpreted under the chosen units/precision
- (-) Handling of grouping and ordering of sheets remains cumbersome to model
- (-) The set of geometrical constructs needs to be extended.

Evaluating Performance



- Benchmarks for “derive" (D) and “optimize” (O) scenarios
- Solving performance is generally good, but also unpredictable (O4 and O8 timed out)

Evaluating Performance



- Solving time increases when list upper bound increases
- Assisting search with domain-specific information might improve performance

Lessons Learned

- Spoofax and MiniZinc took care of much of the “heavy lifting” in realizing CSX.
- A systematic approach to DSL evaluation is useful for communicating about a DSL in an industrial context.
- Starting to use a DSL in practice has a big impact on the software engineering process.
- The conceptual power of CSX is amplified by its IDE.
- It is a crucial language design decision to have types being defined in a language itself.
- Switching to the constraint-based programming paradigm can be challenging for developers that have no experience with constraint programming or with declarative programming at all.

Future Work

- Further extend coverage gaps
- Apply in practice, with domain experts, possibly mechanical engineers
- Use domain specific information to improve performance

Taming Complexity of Industrial Printing Systems Using a Constraint-Based DSL

An Industrial Experience Report

LangDev'22 - September 28, 2022 - Aachen

Jasper Denkers - j.denkers@tudelft.nl,
Marvin Brunner, Louis van Gool, Jurgen Vinju, Andy Zaidman, Eelco Visser

Deployment Architecture