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
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 print products and production options for a given product, and (3) the large range of finishers produced by different vendors.

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 and finishing 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 large variety of printing systems.

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, coping with the large variety of printing systems; besides devices that produce books, there are many others that, e.g., produce magazines, packaging, or decoration.

Constraint solving seems 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

```
1 type Sheet { width: int, height: int }
2 type Stack { sheets: list<Sheet> }
3 device D {
4     location a: Stack
5 }
```

(a) CSX model of device D that instantiates the user-defined record-type Stack (modelled as list of Sheets) in location a.

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

(b) MiniZinc [14] constraint model for device D with variables for the size and properties of the stack's sheet list, for an upper bound of 10 sheets. The constraints on lines 3–4 frame variables that are not considered in the sheet list to a default value (0 for integers).

```
1 a = Stack([
2   Sheet(2100, 2970), Sheet(2100, 2970)
3 ])
```

(c) A configuration for device D based on the SMT solution of (d).

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

(d) A solution found by an SMT constraint solver that corresponds to two sheets of width 2100 and height 2970.

Figure 1. An artificial CSX model (a), its translation to constraints (b), a solution for the constraint model (d), and the solution mapped back to a configuration on the CSX level (c).

issue. With a DSL, we can automate the transformation of more high-level models of printing systems to constraint models. On these generated constraint models, we use constraint solvers to find (optimal) configurations and realize automatic configuration space exploration. The modelling of printing systems in the DSL is in terms of the printing domain and abstracts over low-level and repetitive constraint modelling, making the modelling task feasible in practice.

We have additional motivations for using a DSL in our context, in contrast to using a GPL. First, a DSL can enable domain experts such as mechanical engineers to contribute to the modelling process. Second, the use of a DSL promises to improve productivity by reducing the turnaround time for developing control software. Third, a DSL can better handle the variability when modelling many similar devices. Finally, a DSL can be accompanied by an IDE specific to its domain, potentially improving usability of the modelling environment.

In previous work, we have developed CSX (Configuration Space eXploration) [5], a DSL for modelling digital printing systems, automatically generating constraint models from device models, mapping solutions back to the domain of printing configurations, and deriving environments for configuration space exploration. Figure 1 depicts the translation of CSX to constraints and the mapping of a solution to a device configuration for an example model. Our hypothesis is that CSX is an effective and scalable method in creating control software for digital printing systems. With sufficient coverage and practical solving performance, it has the potential to improve the current state of control software development for printing systems by adding functionality (introducing configuration space exploration that is automatic and complete) and reducing software engineering complexity.

In addition to validating the concepts of CSX, our objective is to evaluate CSX's practical applicability. This has guided our approach in two ways. First, we design the language from the perspective of the user, top-down, meaning that we do not restrict language features before having substantiation for such restrictions from practice. Second, we use MiniZinc [14] as a facade for different underlying SMT solvers [1], as we did not want an early design decision for a specific solver to later hinder our experimentation opportunities.

Although CSX 1.0 was already suitable for modelling devices and realizing configuration space exploration for realistic scenarios, empirical results have shown that it does not yet effectively cover all aspects of the full range of digital printing systems. In particular, we identify and tackle the three most prominent problems of CSX 1.0, that if solved, would bring CSX closer to applicability in practice: (1) CSX 1.0 is limited to modelling a stack of sheets uniformly. To allow more detail in models, we need to be able to model sheets in stacks individually. For that, we add support for generic lists in CSX 2.0. (2) Geometrical concepts such as orientations and transformations are heavily used in printing systems, but require modelling on a low level of abstraction in CSX 1.0. To effectively incorporate geometrical aspects in models, we add geometrical constructs to CSX 2.0 that abstract over linear algebra. (3) CSX is a constraint-based language and therefore involves a style of modelling that can be unintuitive for software engineers that are not familiar with constraint programming. Therefore, in CSX 2.0 we add support for operators in the style of functional programming that are automatically translated to constraint-based counterparts.

In summary, our contributions with respect to our previous work on CSX 1.0 are as follows:

- CSX 2.0, which adds language support for generic lists, geometrical constructs, and functional-style operators.
- An evaluation of CSX 2.0 in an industrial context.

- Lessons learned on using a constraint-based DSL in an industrial context.

2 Industrial Printing and Finishing Systems

2.1 Printing and Finishing

Digital printing systems consist of a printer and finishing equipment where the printer prints individual sheets and the finishing equipment handles subsequent processing steps. Examples of finishing devices are an edge stitcher and a booklet maker. An edge stitcher takes a stack of sheets and binds them by stitching one or more stitches at an edge. A booklet maker takes a set of individual sheets as input and produces a booklet as output by stitching, folding, and trimming. Ideally, print system end-users (e.g., operators in a print shop) can operate the printing system as a whole, in which printing and finishing are fully integrated.

Although finishing devices are capable of processing large volumes of printing products at high productivity, they have mechanical, hardware and software limitations that influence their configuration possibilities. The challenge of an operator that uses such devices is: given the available input materials and printer capabilities, how do I need to configure the finishers such that I obtain the desired end product? Answering this question is an exercise in configuration space exploration: finding a complete configuration that is possible with the devices at hand and that leads to the manufacturing of a product that satisfies the client's wishes. Even for a seemingly simple device such as an edge stitcher, reasoning about its configuration space can already become complex.

As an example, we take an edge stitching device such as depicted in Figure 2. This device takes a stack of sheets of limited sizes, stitches the stack at the right edge, and optionally rotates the stitched stack before outputting it. Table 1 depicts four scenarios of configuration space exploration for this device.

Scenario A considers as input a stack of A4 sheets in portrait orientation without rotation after stitching. We can compute a complete configuration for this scenario stepwise from input to output. At the location Stitched, the stack of sheets is still in A4 in portrait orientation, but with a stitch at the right edge. Since we do not rotate, the output location gets the same characteristics.

Scenario B is more complicated, as it requires an *output* of A4 sheets in portrait orientation with the stitch at the top edge. We need to reason from output back to input to find a configuration for this scenario. The scenario is possible, and requires to take A4 sheets in landscape orientation as input with a rotation of 90 degrees after stitching. Similarly, we can derive a configuration for A3 sheets in portrait orientation with the stitch at the top edge (scenario C).

Scenario D is not possible. It requires A3 sheets in landscape orientation with the stitch at the top edge. The stitch

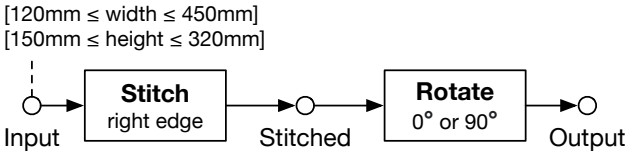


Figure 2. Schematic overview of the finishing steps of a simplified edge stitching device. Dots indicate locations at which we consider a snapshot of the stack of sheets that is stitched. The input is limited to the sizes of sheets it can handle. Rectangles represent the actions that are performed on the objects. The device can only perform a right-edge stitch, and can subsequently rotate the stack by 0 or 90 degrees.

Table 1. Scenarios of configuration space exploration for the edge stitching device from Figure 2. Each row (A-D) represents a scenario. The middle four columns indicate the values for a configuration corresponding to the scenario. Question marks (?) indicate unknown values, for information that needs to be derived by configuration space exploration. Width (w), height (h), and stitch edge (e) are abbreviated and the millimeter unit is ommitted.

Sc.	Input	Stitched	Rotation	Output	Possible
A	w=210 h=297	?	0°	?	yes
B	?	?	?	w=210 h=297 e=top	yes
C	?	?	?	w=297 h=420 e=top	yes
D	?	?	?	w=420 h=297 e=top	no

at the top edge requires to rotate 90 degrees after stitching, because the device is limited to stitching at the right edge. The rotation implies that the input for this scenario should be A3 sheets in portrait orientation. However, A3 sheets in portrait orientation, with a height of 420mm, violates the device's input size limitation of maximum height 320mm.

The context of our work is Canon Production Printing. This company develops and manufactures printers which need to be integrated with finishers from many external vendors. Therefore, we are mostly concerned about the configuration spaces of *finishing* devices, and realizing control software for integrated systems of printers and finishers.

In principle, we can use a general purpose programming language to model printing systems, and to implement algorithms for finding configurations for the finishing devices. However, using a general purpose programming language

has two problems. First, the systems span large configuration spaces, in which configurations for print jobs need to be found automatically — taking all the features and limitations of the devices into account. Second, the variety of printing systems is large, involving many variants that can behave similarly, but have subtle differences. These problems make developing and maintaining control software for printing systems complex.

A natural starting point for modelling a printing system, or a manufacturing system in general, is to identify locations through which objects of the manufacturing process pass. Next, we can define each action with its parameters that alter the manufacturing objects from one intermediate location into the next. By doing so, each location represents a snapshot of the process that transforms the input step-wise. These actions and locations correspond to those in Figure 2.

Based on these snapshots of the manufacturing process and the actions that occur in-between these snapshots, we can use *simulation* to find a configuration. Simulation means that we start with a given input object at the first snapshot location of the model and calculate consecutive snapshots by executing actions for their given parameters. Note that this is an imperative approach: the input and parameters need to be known up front and we can then calculate the end result step by step. Also, by calculating the manufacturing objects at each snapshot location of the device, we can check whether the (partial) configuration — consisting of the input objects and action parameters — conforms to the device's limitations. If device limitations are violated, the printing system operator needs to try again with an updated specification of the input and the action parameters. If we would want to go the other way around, e.g., by requesting a desired end result, the simulation approach falls short in finding the corresponding input objects and action parameters.

The existing control software at Canon Production Printing is based on the aforementioned simulation approach, which *constructs* configurations. To partially overcome the inherent limitations in configuration space exploration of a constructive algorithm, heuristics are added that automatically derive partial configurations for particular cases of output product decriptions. In the software implementations, these heuristics still not cover the complete configuration space. There remain configurations that the device can handle but the control software cannot derive. Moreover, the heuristics are device-specific and not composable, and therefore hinder reusability and maintainability. In the rest of this paper, we refer to this approach as *pre-CSX*.

We need an *analytical* approach — different than the constructive approach based on simulation and manual heuristics — that automatically derives the complete configuration space of a device, given only a partial description of a configuration. Because the configuration space is large and actions are inter-dependent, this is hard to achieve with manual programming effort, especially given the large variety of printing systems. This is where we can leverage the power of constraint solving, which seems a natural fit for configuration space exploration. By specifying the configuration spaces in terms of constraints, we can use constraint solvers to find configurations. It does not matter anymore for which scenario — forward, backward, or anything in between such as finding parameters given an input and output. The problem with modelling digital printing systems directly in a generic constraint modelling language, however, is that it is tedious and repetitive work.

2.2 Requirements

Our objective is to obtain a method for modelling printing systems and deriving environments for automatic configuration space exploration that satisfies the following requirements:

Domain Coverage The modelling language covers the aspects and features of digital printing systems.

Configuration Accuracy The automatic finding of configurations for said aspects and features is *correct* (configurations that are found conform to the device's limitations; there are no false positives) and *complete* (i.e., there are no configurations that are not found but that are possible on the device; there are no false negatives).

Configuration Performance Configurations are found in the order of seconds, i.e., in a timespan that is considered practical by control software engineers for use in interactive UIs.

2.3 CSX: Configuration Space eXploration

Previously, we have developed CSX [5]: a language and environment that serves as an interface to constraint programming specific to the printing domain, abstracting over the complexity of developing control software in two ways. First, CSX offers domain-specific constructs that abstract over low-level details. Such details do not need to be rethought each time a new device is modelled. In CSX, a library of actions can be built, which can be reused in device models. Second, by leveraging the power of constraint solvers to find configurations for printing devices, control software engineers do not have to manually develop algorithms to find configurations. Therefore, CSX promises to tackle two of the most challenging aspects of developing control software for printing systems.

In CSX, we model printing systems by modelling intermediate locations of the manufacturing process and configuration parametricity. We will now further introduce CSX's language concepts using Figure 3, an example CSX model of an edge stitching device.

User-defined record-types. In CSX, we use *user-defined record-types* to model the objects in the manufacturing process. This concerns the input, output, and snapshots of the products at intermediate locations. Instead of specifying

```
1  type Sheet {
2    width: int,  [width > 0],
3    height: int, [height > 0]
4    // Width and height in 1/10mm precision
5  }
6
7  type StitchedStack {
8    sheets: list<Sheet>,
9    stitchEdge: edge
10 }
11
12 action Stitcher(input: list<Sheet>,
13                 output: StitchedStack) {
14   // At least two sheets required for stitching
15   [size(input) ≥ 2]
16
17   parameter stitchEdge: edge
18
19   [output.sheets == input]
20   [output.stitchEdge == stitchEdge]
21 }
22
23 device EdgeStitcher {
24   location input: list<Sheet>
25
26   [size(input) ≤ 10] // Max number of sheets
27
28   [input.forall { sheet =>
29     // Min and max sheet sizes
30     sheet.width ≥ 1200 and sheet.height ≥ 1500 and
31     sheet.width ≤ 4500 and sheet.height ≤ 3200
32   }]
33
34   stitcher = Stitcher(input, stitched)
35   // This device can only stitch on the right edge
36   [stitcher.stitchEdge == right]
37
38   location stitched: StitchedStack
39
40   parameter rotation: orientation
41   [rotation == rot0 or rotation == rot90]
42
43   [output.sheets.forall { sheet =>
44    (rotation == rot0 implies sheet == stitched.sheets[index])
45    and
46    (rotation == rot90 implies (
47      // Swap width and height in case of 90 degrees rotation
48      sheet.width  == stitched.sheets[index].height and
49      sheet.height == stitched.sheets[index].width
50    ))
51   }]
52   [size(output.sheets) == size(stitched.sheets)]
53
54   [output.stitchEdge ==
55     orientate(stitched.stitchEdge, rotation)]
56
57   location output: StitchedStack
58 }
```

Figure 3. CSX model of an edge stitching device (schematically depicted in Figure 2) that has a list of sheets as input, stitches them in the right edge, and optionally rotates the stitched stack 90 degrees before it leaves the machine as output. Integer dimensions in this model represent 1/10mm.

all properties individually, CSX users can define record-like types such as sheets and stacks. In the example, `Sheet` and `StitchedStack` are user-defined types (lines 1–9). User-defined types are records of properties which can be either defining properties or derived properties. We can use this abstraction to incorporate objects such as sheets and stacks in a device model to model the snapshots of printing products. Besides user-defined types, the language supports integers and booleans as primitives.

Actions. Units of printing behavior are captured in *actions*. Actions are defined for one or more locations that can be inputs or outputs. In the example, the stitching behavior is captured in an action (lines 12–21). Additionally, actions can contain parameters that contribute to the configuration space, such as the `stitchEdge` parameter (line 17).

Devices & Locations. We can model *devices* in CSX, which are representations of systems that instantiate the user-defined types for snapshot products at *locations* and instantiate actions in between those snapshots. The example model considers three snapshot locations of the printing objects: `input`, `stitched`, and `output`. The configuration space of a device is defined as the possible values for all locations and action parameters that conform to the constraints. CSX supports modular decomposition in the sense that devices are modelled by building on a set of reusable type and action definitions, which can be instantiated in varying ways.

Constraints. In square brackets, we can write expressions to form *constraints* that limit the configuration space of a device. Examples of constraints are enforcing that sheets have a positive width and height (lines 2–3), stitching requires at least two sheets (line 15), and the minimum and maximum sheet sizes the device can handle (lines 28–32). Additionally, constraints express how snapshot printing objects relate to other snapshot printing objects in the device. For example, the rotation parameter impacts whether the width and height of sheets are swapped between the `stitched` and `output` location.

Scenarios & Tests. In CSX models, we can also define *tests* for devices. Such tests are used to specifiy configuration space exploration scenarios with assertions to validate the models. Figure 4 lists several tests for the example model. By using *scenarios*, a restricted configuration space can be considered for multiple tests nested in the scenario. The assertions can simply expect there to be a configuration (`succeeds`), no configuration (`fails`), or expect something more specific using the constraint notation. For example, the first test expects a rotation of 0 degrees (line 14) and the second test expects rotation by 90 degrees (line 19).

We have implemented the CSX language and IDE using the Spoofax language workbench [10]. We express the translations of CSX models to SMT models in MiniZinc [14]. MiniZinc is a generic and solver-independent constraint modelling language, which allows us to use various solvers for finding configurations. Both the translation of CSX models and tests

```
// All tests are for the EdgeStitcher device and
// with two sheets in the output
scenario device EdgeStitcher
         config size(output.sheets) == 2 {

  // A4 sheets in portrait orientation
  scenario 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 right edge requires no rotation
    test config output.stitchEdge == right {
      [rotation == rot0]
    }

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

  }

  // 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
  }

  // A3 sheets in landscape orientation with stitch on
  // right edge is possible
  test config output.sheets[1].width  == 4200
       config output.sheets[1].height == 2970
       config output.sheets[2].width  == 4200
       config output.sheets[2].height == 2970
       config output.stitchEdge == right {
    succeeds
  }

}
```

Figure 4. Tests accompanying the CSX model of Figure 3.

to MiniZinc and the mapping of SMT solutions back to the CSX-level are implemented using the Stratego transformation language [2]. Tests are evaluated interactively in the IDE, i.e., a test is re-evaluated automatically if and only if it or the device under test has changed. Figure 5 depicts how feedback on tests is presented in the IDE.

Figure 6 depicts an architecture that applies CSX to realize control software. We have implemented the CSX language and IDE and all components relevant for automatic configuration space exploration. The deployment of CSX with code generation for communication with embedded software in devices and the integration with user interfaces is future work. For further details about the semantics and implementation of CSX 1.0, we refer to our previous work [5]. The

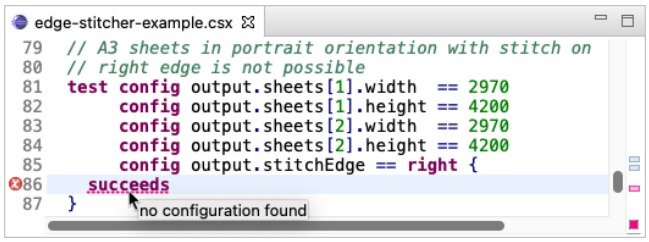


Figure 5. An example of interactive validation in CSX. The test (a modified version of the third test in Figure 4) incorrectly expects CSX to find a configuration. The CSX IDE reports that this expectation is incorrect using an error marker. While hovering over the incorrect expectation, the popup indicates that no configuration was found.

technical contribution of the current paper focusses on extending the coverage of CSX within the existing framework, which we discuss next.

2.4 Coverage Gaps

For a domain-specific language such as CSX to be successful, it is crucial that the language's constructs are adequate in covering the printing domain. Although CSX 1.0 was found to be suitable in modelling realistic printing systems and to realize configuration space exploration with practical performance [5], further application of the language on more printing systems revealed limitations of the approach. We discuss three coverage gaps in this section, for which we extend the language in the following section.

Non-Uniformity. In CSX 1.0 one is limited to modelling uniform stacks of sheets. In case of a booklet maker, one would be limited to only modelling booklets with a uniform stack of sheets. If we would like the cover sheet to be of a different type, the cover sheet needs to be modelled separate from the body sheets. Non-uniformity can be dealt with in a more generic way using lists of sheets. In the example, we have used the newly introduced generic lists to model non-uniform stacks. A list is, e.g., also useful for a stitching device that can stitch with a variable number of stitches.

Geometry. Modelling geometric transformations requires low-level modelling in CSX 1.0. For example, one could manually define constraints for each case a rotation parameter can take, or manually implement linear algebra. In the example, we have used the newly introduced geometrical constructions (e.g., for edge) and transformations.

Function-Style Operators. Operators in a CSX 1.0 are expressed using predicates. This predicate-style operators do not naturally express processing steps of printing processes which are directional. Functional-style operators do express a direction and therefore they are more appropriate for modelling printing systems. In the example, `orientate` is an example of an operator in functional style.

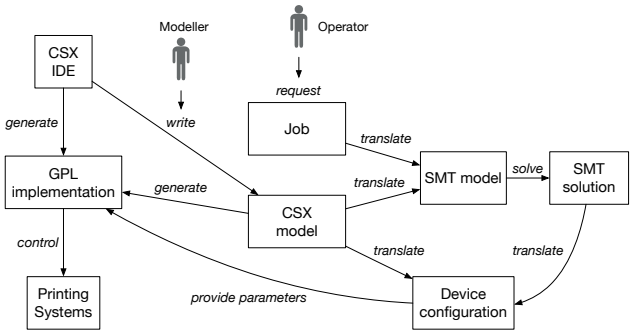


Figure 6. An architecture in which CSX is applied to realize control software for digital printing systems. GPL stands for a general purpose programming language such as C#.

3 Increasing Domain Coverage

In this section, we describe how we have engineered CSX 2.0, that features improved coverage of the digital printing domain.

3.1 Non-Uniform Stacks of Sheets

Although the existing version of CSX was able to cover useful printing systems, it lacked the ability to model non-uniform stacks of sheets. To support non-uniform modelling, we need an additional data structure for generic and ordered collections. For this purpose, we add support for lists to CSX 2.0.

In the configuration space of devices such as booklet makers, products with a variable number of sheets can be produced. Therefore, we need to be able to model devices with lists that have a variable size. Implementing a list with a variable size is trivial in object-oriented programming. When a list needs to grow, additional memory can be allocated for this list at runtime. However, in constraint programming with modelling languages such as MiniZinc, realizing this is not trivial. A constraint model needs to define all variables up front; additional variables cannot be added at runtime. Lists with a variable size therefore do not naturally map to the constraint domain.

In CSX 2.0, we add support for the generic `list<T>` type for lists with elements of type `T`. Lists in CSX 2.0 do have a variable size, and can be instantiated for both primitive and user-defined types. Figure 1a shows an example that models a stack as a list of sheets (line 2).

We realize variably sized lists by using fixed size arrays in the target (MiniZinc) model that are only partially considered in the actual configuration. The size of the target arrays (n_{max}) determines the range of sizes the list on the CSX level can have; the CSX list thus has an upper bound on its dynamic size. When translating the CSX model, a n_{max} should be chosen that ensures the configuration space is sufficient for the particular model.

For a CSX list with values of a primitive type, e.g., a list of integers, a single array is needed in the target model. For a CSX list of a user-defined type such as a sheet with multiple properties, the target model gets an array for each property. Additionally, the target model contains an integer variable that indicates the size n of the list ($0 \leq n \leq n_{max}$) in the configuration.

When mapping solutions found in the constraint domain back to CSX, only the first n elements of the arrays are considered. On the target level, the elements in the array for size positions $n < i \leq n_{max}$ (using 1-based indices) are ignored and framed to default values to avoiding what is commonly called "junk" in constraint solving. By doing so, CSX lists behave as dynamically sized lists. Although this approach could also work for nested lists by adding extra dimensions to the arrays in the generated constraint model, we have not found a need for it in the domain of printing.

Figure 1 shows an example of a stack that is modelled as a list of sheets and how that translates to constraints, expressed in MiniZinc. The variable `a_sheets_size` represents the size of the list of sheets. The domain for this variable (`0..10`, i.e. an integer value in the range 0 to 10) represents the possible sizes of the list (given that the upper bound n_{max} is 10). Sheets have two properties: `width` and `height`, both of type `int`. The target model contains an array with variables for each property, denoted in MiniZinc with, e.g., `array [1..10] of var int : a_sheets_width` for the `width` property. Again, the size of this array is determined by the upper bound on the size of lists.

The `forall` constraint makes sure that the elements in the arrays that are not part of the actual solution (i.e., for indices larger than the size of the list), are set to a default value. This enforces that a single CSX configuration corresponds to a single solution at the constraint level. Otherwise, multiple solutions at the constraint level could correspond to the same configuration at the CSX level, making the solution space unnecessarily large. Note, however, that CSX does not prevent references to values outside the size of the list.

To make lists in CSX practical, we add operations on lists such as `reverse`, `append`, `prepend`, and `map`. Many of such expressions over lists can be expressed by translation into a `forall` construct. The `forall` and `exists` quantifiers are common constructs in (functional) languages that support lists. The `forall` operator is used to express whether a predicate holds for all elements in a collection. We implement the `forall` construct and use it as a core construct which other operations translate to, see Figure 7.

Note that the `index` operator in the examples is implicit and can only be accessed in the context of a `forall`. It denotes the 1-based index of the element in the list for which the predicate is declared. By design, the `forall` operators cannot be nested, because for a single `index` operator it would not be clear to which forall it corresponds. Although a

```
var xs: list<int>
var ys: list<int>

reverse xs into ys
```

```
size(ys) == size(xs)
xs.forall { x =>
  ys[size(xs) + 1 - index] = x
}
```

(a) `reverse`

```
var xs: list<int>
var ys: list<int>

map xs with x => x + 1 into ys
```

```
size(ys) == size(xs)
xs.forall { x =>
  ys[index] = x + 1
}
```

(b) `map`

```
var xs: list<int>
var ys: list<int>
var z: int

append z after xs into ys
```

```
size(ys) == size(xs) + 1
xs.forall { x =>
  ys[index] = x
}
last(ys) == z
```

(c) `append`

```
var xs: list<int>
var ys: list<int>
var z: int

prepend z before xs into ys
```

```
size(ys) == size(xs) + 1
first(ys) == z
xs.forall { x =>
  ys[index + 1] = x
}
```

(d) `prepend`

Figure 7. Translation of CSX's list operations (left-hand sides) into `forall` (right-hand sides, also CSX). The target models (right-hand sides) have variable declarations omitted for brevity.

specialized `index` operator would be possible, we think this would make the language unnecessarily more complex.

In addition to the `forall` construct, we add support for list access using square brackets. The `first` and `last` functions are implemented by translating to list access. The `size` function translates to the variable on the constraint level that represents the list size.

We call the operations in Figure 7 to be in *predicate-style*, i.e., the operators enforce a predicate over multiple (list) variables. For example, `reverse xs into ys` evaluates to true if *xs* is the reverse of *ys*. The type of the predicate-style `reverse`, `append`, `prepend`, and `map` operations is therefore boolean.

Many aspects in printing processes are directional, which are unnatural to capture in constraint programming or with predicate-style operators. It would be more natural to be able to use list operations in *functional-style*, in which the result of the operations is also a list. Similar to functional programming, that would allow chaining of operations, i.e., combine operators in such a way that the output of one operator is directly considered as input to the next operator. When modelling a printing system, such operators better capture the actual direction of the manufacturing process.

While a functional language would dynamically allocate memory for intermediate values of such chains of operations at runtime, in constraint programming the variables need to be known up front. In Section 3.3 we describe an algorithm for introducing intermediate variables where needed. That will allow writing, e.g., `ys == reverse(xs)`, which will then first translate into the predicate-style variant (`reverse xs into ys`), which in turn will translate into a `forall` expression.

3.2 Geometrical Constructs

Concepts from the geometrical domain such as orientations, transformations, and edges are frequently used in the printing domain. For example, finishing devices can have the possibility to orientate input sheets to be flexible in input and output formats. A hardware characteristic might limit the maximum width of a sheet in a location in the manufacturing process. By being able to rotate the sheet after such a location, there are more possibilities for sheet sizes in the following steps.

Typically, geometrical properties and transformations are captured numerically, in linear algebra. Transformations of orientations or edges are then expressed by matrix multiplication. Although matrices could be expressed using user-defined types in CSX with properties for the matrix elements, it involves modelling on a low level of abstraction. By lifting a restricted but high-level set of geometrical constructs from the numerical domain to a symbolic domain in CSX 2.0, incorporating geometrical aspects in models can be done at a high level of abstraction.

Although arbitrary transformations could be expressed using matrices, many transformations in the printing domain are limited to rotation over a multiple of 90 degrees, either with or without flipping. We reflect this in CSX 2.0 by including a restricted set of orientations that correspond to those commonly used transformations: `rot0`, `rot90`, `rot180`, `rot270`, `flip0`, `flip90`, `flip180`, and `flip270`. In the constraint model, those orientations correspond to a 2-by-2 matrix. For example, `rot90` corresponds to $\left[\begin{smallmatrix} 0 & 1 \\ -1 & 0 \end{smallmatrix}\right]$.

To effectively use orientations, we introduce high-level constructs for concepts that can be orientated such as edges. Again, edges could already be modelled using primitives or user-defined types, but having dedicated constructs enables us to implement concise operations for them with orientations. An edge is one of `top`, `right`, `bottom`, or `left`, which

are represented as two-dimensional vectors in the constraint domain. For example, `top` corresponds to $[\,0\ 1\,]$.

We translate the application of an orientation to an edge in the constraint model to matrix multiplication. As an example, we take the rotation of the top edge over 90 degrees. In CSX 2.0, we can express this using `e == orientate(top, rot90)`, in which variable `e` of type edge is considered equal to the result of the rotation. At the constraint level, this would correspond to the matrix multiplication $[\,0\ 1\,] \times \left[\begin{smallmatrix} 0 & 1 \\ -1 & 0 \end{smallmatrix}\right] = [\,-1\ 0\,]$. We interpret the resulting vector $[\,-1\ 0\,]$ as the `left` edge such that the linear algebra is hidden from the CSX user.

Figure 8 depicts an artifical CSX model and corresponding MiniZinc for a device that transforms an edge over an orientation. The translation of geometrical constructs in CSX to MiniZinc makes use of a prelude. This is MiniZinc code that represents data types, predicates, and functions over the geometrical constructs which are referenced in the target (MiniZinc) model of a specific CSX model.

An edge is represented with a two-dimensional vector in MiniZinc, using the type `array[1..2] of var -1..1`; an array with indices in the range `1..2` and with values in the domain of `-1..1`. Since variables of this type can get values that do not correspond to one of the four edges we consider (e.g., $[\,1\ 1\,]$ does not represent one of the four edges), we need to frame its instances. We do so by applying the `isEdge` predicate to each instance. For each instance of an edge variable in CSX, a two-dimensional array is declared on which the `isEdge` predicate is applied. We realize orientations in a similar way.

CSX supports high-level operators for geometrical constructs. A CSX user does not need to write out matrix multiplications, but can directly express operations on the geometric data structures using these operators. For example, one could write `e2 == orientate(e1, o)`. Type checking ensures that only valid combinations can be used for transformations.

Using constraints in the backend of CSX involves a translation that is bidirectional. The constructs in CSX are translated to variables and constraints in the constraint domain. Also, solutions in the constraint domain are mapped back to the CSX level. Interpreting a solution for a geometrical construct involves a new mechanism. For example, for orientations and edges we need to map the individual values that are found in the solution to one of the possibly restricted values on the CSX domain. If for an edge in the constraint domain the value $[\,0\ 1\,]$ is found, that would map to the `top` value at the CSX level. Orientations are interpreted analogously.

We also add support for lists of orientations and edges. The arrays for orientations and edges in the constraint model then get an extra dimension.

```
// Artificial device model that transforms an edge (in CSX)
device D {
  location e1: edge
  location e2: edge
  location o: orientation

  [e2 == orientate(e1, o)]
}
test device D
     config e1 == top
     config e2 == bottom {
  [o == rot180 or o == flip0]
}
```

```
% Prelude for target model (in MiniZinc)

% Constants for the eight relevant orientations
array[1..2,1..2] of var -1..1: Rot0    = [| 1, 0| 0, 1|];
array[1..2,1..2] of var -1..1: Rot90   = [| 0, 1|-1, 0|];
array[1..2,1..2] of var -1..1: Rot180  = [|-1, 0| 0,-1|];
array[1..2,1..2] of var -1..1: Rot270  = [| 0,-1| 1, 0|];
array[1..2,1..2] of var -1..1: Flip0   = [| 1, 0| 0,-1|];
array[1..2,1..2] of var -1..1: Flip90  = [| 0,-1|-1, 0|];
array[1..2,1..2] of var -1..1: Flip180 = [|-1, 0| 0, 1|];
array[1..2,1..2] of var -1..1: Flip270 = [| 0, 1| 1, 0|];

% Constants for the four relevant edges
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 to restrict an orientation's matrix
predicate isOrientation(array[1..2,1..2] of var -1..1: o) =
  o = Rot0  ∨ o = Rot90  ∨ o = Rot180  ∨ o = Rot270 ∨
  o = Flip0 ∨ o = Flip90 ∨ o = Flip180 ∨ o = Flip270;

% Predicate to restrict an edges's matrix
predicate isEdge(array[1..2] of var -1..1: e) =
  e = Top ∨ e = Right ∨ e = Bottom ∨ e = Left

% Function for orientating an edge
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] + e[2] * o[2,2]
  ]);
```

```
% Device-specific target model (in MiniZinc)
array [1..2] of var  -1..1 : e1;
constraint isEdge(e1);
array [1..2] of var  -1..1 : e2;
constraint isEdge(e2);
array [1..2,1..2] of var  -1..1 : o;
constraint isOrientation(o);
constraint e2 == orientateEdge(e1,o)
```

Figure 8. Top: An artificial CSX model with an orientation parameter `o` that transforms edge `e1` into `e2`. Middle: The prelude MiniZinc code for geometrical constructs that is added to target models. Bottom: The device-specific target model, making use of the prelude.

3.3 Functional-Style Operators

A printing process is typically directional: input objects are processed step by step into output objects. The functional paradigm supports modelling such a directional process naturally. Functional-style operators are composable and thus can be chained. By doing so, a sequence of chained operations expresses an order or direction in computation — similar to the order of manufacturing steps that are involved in a printing system.

For atomic values such as integers, booleans, and user-defined enums, chaining of operators is supported by default in MiniZinc and therefore also in CSX. Such atomic values are represented by a single variable in the constraint domain. In contrast, compound values such as user-defined types, lists, and geometrical constructs, are represented by multiple variables in the constraint domain. Chaining of operators on compound values is not supported in MiniZinc. Still, we want to support chaining of operations in CSX on compound values, too, as it would make the switch from functional programming languages to CSX easier. Therefore, in addition to the predicate-style operations on, e.g., lists and geometric constructs, we add functional variants that support the chaining of such operations.

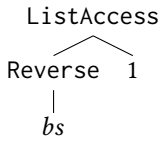
An expression written in predicate-style such as `reverse x in y` could be written in functional notation as `y == reverse(x)`. More interestingly, an expression such as `reverse x into y and z == y[2]` could be written as `z == reverse(x)[2]`. To achieve such style of modelling — that allows chaining of operations — we need to the instantiate constraint variables for the intermediate results.

Imposing the responsibility for declaring the intermediate variables to the language user would negatively impact CSX's usability. To prevent this, CSX derives where intermediate variables are necessary and introduces them automatically. CSX 2.0 analyses expressions to detect and unfold chained operations where necessary. In case of chained operations, intermediate variables are introduced and the expressions are rewritten in a form that makes use of the intermediate variables and that removes the chaining. This happens in an intermediate translation step in the CSX compilation pipeline.

Let's take the following CSX spec:

```
1 var bs: list<bool>
2
3 [reverse(bs)[1]]
```

The constraint `reverse(bs)[1]` has the following abstract syntax tree:



In this tree, the `Reverse` node represents a compound value that is derived from the `bs` variable, and it is an input for the `ListAccess` node, which also derives a new compound value. Therefore, this expression requires an intermediate variable to be inserted.

It will be translated into:

```
1 var bs: list<bool>
2 var i1: list<bool>
3
4 [reverse(bs) == i1]
5 [i1[1]]
```

A new variable `i1` is introduced which is considered equal to the reverse of `bs`. Then, the initial expression gets expressed in terms of the new variable.

Still, the deriving reverse expression is not in predicate-style. Since it is on one side of an equals expression with an instance value on the other side, we can rewrite it into predicate-style. This results in:

```
1 var bs: list<bool>
2 var i1: list<bool>
3
4 [reverse bs into i1]
5 [i1[1]]
```

Finally, the transformation step for predicate-style operations on lists (as defined in Section 3.1) transforms the expression into a `forall` construct:

```
1 var bs: list<bool>
2 var i1: list<bool>
3
4 [bs.size == i1.size]
5 [bs.forall { x => i1[bs.size + 1 - index] = x }]
6 [i1[1]]
```

Algorithm. The algorithm repeatedly finds and rewrites expressions from functional-style into predicate-style until no functional-style operators remain. The algorithm starts with identifying the types of variables in an expression abstract syntax tree. First, it identifies the references of locations, parameters, and other variables as *instance values*. Instance values are references to explicitly declared variables in a CSX model. Second, the nodes for operations on compound data types are identified as *derived values*. Derived values are the result of an operation on another, possibly compound, value and might require the introduction of an intermediate variable.

The algorithm repeatedly tries to find and replace operations that take a derived value as an input and have a new derived value as output. For such cases, the derived base value needs to be replaced by a newly introduced intermediate variable. When this process finishes, i.e., there are no nodes left that need an intermediate variable, we can start rewriting functional operations to predicate-style operations. Finally, no functional-style operations are left, and the normal form with only predicate-style operations remains.

4 Industrial Evaluation

Having designed the CSX language and associated environment for configuration space exploration, we now evaluate the industrial application of CSX 2.0 at Canon Production Printing. We explore whether CSX as a constraint-based language is effective for modelling printing systems and realizing automatic configuration space exploration. We evaluate whether CSX meets our requirements (Section 2.2) on domain coverage, configuration accuracy, and configuration performance. Additionally, we evaluate the relevance of the approach. In particular, we consider the following evaluation questions:

Domain Coverage Does CSX provide the constructs for modelling devices at Canon Production Printing, without having to resort to low-level constraint programming?

Configuration Accuracy Is configuration space exploration with CSX accurate, i.e., is it *correct* (configurations that are found conform the device's limitations; there are no false positives) and *complete* (there are no configurations that are not found but that are possible in the device; there no false negatives)?

Configuration Performance Do the generated constraint models in MiniZinc find (optimal) solutions in reasonable time (within seconds) for realistic printing system models?

Relevance Is it *sufficient* to use CSX to achieve automatic configuration space exploration and is it *necessary* to use CSX instead of directly modelling printing systems in a generic constraint modelling language such as MiniZinc?

In the remainder of this section, we describe our evaluation method and discuss results per question. Although this paper presents an extension of an existing version of CSX, we evaluate CSX 2.0 as a whole — not only the new features.

4.1 Domain Coverage

Study setup. Two of the authors participate in an exploratory case study in which a realistic printing device is modelled in-depth from scratch. The industrial context of the study is Canon Production Printing and we consider a case for which the company has also developed control software in practice. The first participant is the developer of CSX. The second participant is a domain expert from Canon Production Printing having 10+ years of experience with developing control software for printing systems. The first participant was the main implementer of CSX. The second participant has been actively involved in the development of the language and environment.

The subject case of the study is a setup including a printer with two input trays, an edge stitcher, and a virtual reader. The input of the device consists of two input trays. Each input tray contains sheets that are the same. Consequently, the stack that is gathered from those input trays can contain at most two different types of sheets, which can occur in any amount and order. Although this seems like a simple case, it was already difficult to cover in the pre-CSX situation.

In the pre-CSX approach, modelling this device with support for only simulation — calculating the output based on the input and parameters — is considered straightforward by the domain expert. For the simulation implementation, the input objects are specified up front and the output is calculated by processing each step, given parameters such as orientations. However, reasoning backwards, e.g., to calculate the configuration parameters given an input and a desired end product, is considered complex by the domain expert. In particular, the freedom on orientation before and after printing results in many configuration possibilities, for which it is not clear how to find all of them. Therefore, this is a relevant case for our study.

The study consists of two parts. In the first part — a think aloud study — the participants model the device in think aloud co-design sessions of two hours. In the second part — reflection analysis — the participants discuss the evaluation questions in a single two-hour session, reflecting back on the co-design sessions using the notes gathered in the sessions.

During think aloud co-design sessions [7] the two participants gather data for evaluating the language design. The participants model the edge stitching device by gradually including aspects of increasing complexity, selected by the domain expert. In particular, these sessions follow the following protocol:

- The participants communicate via a video call. The first participant has the CSX IDE open and shares the screen with the other participant, such that both participants can see the IDE and models.
- The participants perform iterations of modelling in sessions of two hours.
- For each iteration, the domain expert selects an aspect of the device to model. Initially, the domain expert selects the most simple aspect of the device. The judgement of the domain expert is leading in gradually expanding the level of detail of the model. For new iterations, the domain expert expands an aspect with more detail or selects a new aspect. Each iteration starts in a new CSX file by copying the previous file, initially starting with an empty file.
- The participants engage in a think aloud conversation [7] on which properties to consider and which design decisions to make during the process. The first participant writes the CSX code that corresponds to the consensus of the participants on how to model the selected aspect.
- The participants document the considerations and design decisions by taking notes in comments of the

CSX code such that the considerations can be revisited when discussing the evaluation questions.
- The participants validate the model by writing tests, and revert to fixing the model if tests reveal flaws in the model.
- The participants repeat this process until the domain expert concludes that the device is modelled with a level of detail that is sufficient for realizing control software.

In the reflection analysis part of the study, the participants discuss the evaluation questions. Per question, the participants reflect on the modelling sessions, revisiting the notes that were documented with comments in the models.

Results. The participants performed six co-design session of two hours. In some sessions, the participants worked on multiple iterations, and some iterations are based on work from multiple sessions. The sessions resulted in seven iterations of CSX models, of which the last contained the final model of the edge stitching device. Table 2 gives an overview of the aspects that were included in each iteration.

We will now discuss the results of the study in more detail. First, we report on general observations from the modelling sessions. Second, we discuss each aspect of the case separately. We report both positive and negative observations. For example, we label the *i*th observation on the aspect of domain coverage with *DOMAIN-COVERAGE i*. We label the *j*th general observation, not related to, e.g., domain coverage specifically, with *GO j*. The models from the session that are included in this section have undergone light editing in order to improve presentation.

General Observations. Before starting on the first CSX model, the participants realized that they should determine the scope of what they will include in the model. The most high-level question in that regard is whether the model should start before or after the printing device. Although CSX has been originally designed with the intention to model and integrate finishing equipment, there is also utility in including part of the printing device in CSX models (*GO 1*).

In particular, a printing device typically has multiple input trays in a component which is called the paper input module (PIM), which determines the number of different types of sheets that can be used as input. It is relevant to include this in the model, as the sheets in input trays are part of the configuration space that is relevant for finishing. Alternatively, we can leave out this part from the CSX model, and consider the output of the printer as input to the finishing device that we model. This output of the printer then potentially can consist of different types of sheets.

In the evaluation, the participants chose to include the input trays of the printing device in the model. The actual printing operation is considered implicit; its effect is not captured in the model in the study. It could be relevant to include the printing in a later iteration, e.g., for modelling the printable area of sheets.

Table 2. The aspects that were included in each iteration of the co-design evaluation sessions. The first attempt at modelling input trays (iteration 3) was incorrect and was modelled again from scratch (iteration 6).

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

The participants found that a convenient first step in every iteration of the modelling sessions was to model the printing objects (sheets and stacks) by adding or extending type definitions (*GO 2*). User-defined types in CSX allowed the participants to be flexible in how the objects that undergo the finishing actions are modelled, similar as in an object-oriented language. The participants noticed that this flexibility is useful for the modelling process that is incremental. They started with simple type definitions and first completed a device model based on these type definitions. Later, they expanded the type definitions to incrementally include more detail (*GO 3*).

Uniform stacks of sheets. In order to start simple and gradually expand the level of detail in the model, the participants chose to start with modelling stacks of sheets that are uniform. This has a restrictive implication: all sheets in a stack are considered equal by design. Although this is an oversimplification which is not realistic, the participants considered it a good starting point.

Figure 9 (lines 1–5), from the first modelling iteration, depicts the type definition for a uniform stack of sheets. Many properties could be included in sheets, but the participants started with a simple representation of sheets with only a size (`width` and `height`). An additional property `sheets` indicates how many sheets are in the stack.

Since properties are of type integer, the participants had to choose a precision (*GO 4*). The participants chose a precision of 1/10mm. This precision is common in the printing domain and considered precise enough. The participants noticed a downside of this approach: a reader of a CSX model does need to interpret integer values with a division or multiplication of 10 when interpreting them in the more intuitive unit of millimeters (*DOMAIN-COVERAGE 1*).

The integer type in CSX has a domain of both positive and negative integer values. Since the size of the sheets and the

```
1  // Precision: 1/10mm
2
3  type Stack {
4    width: int, [width > 0],
5    height: int, [height > 0],
6    sheets: int, [sheets ≥ 0]
7  }
8
9  device PaperInputModule {
10   location input: Stack
11
12   [input.width ≤ 4500 and input.height ≤ 3200] // Max size
13   [input.width ≥ 1200 and input.height ≥ 1500] // Min size
14
15   [output == input]
16
17   location output: Stack
18 }
19
20 // Check that the configuration conforms to the test setup
21 test device PaperInputModule config input.width == 2100
22                 config input.height == 2970
23                 config input.sheets == 1 {
24   [output.width == 2100]
25   [output.height == 2970]
26   [output.sheets == 1]
27 }
28
29 // Check that no configuration can be found for 100x100mm
30 test device PaperInputModule config input.width == 1000
31                              config input.height == 1000 {
32   fails
33 }
```

Figure 9. The CSX model resulting from iteration 1. A simple device that takes a uniform stack with a size (width and height) and number of sheets as input and outputs the same stack. Hardware limitations on the size of the input stack are captured in constraints. Two tests cover a succeeding and failing scenario.

number of sheets cannot be negative, the participants added constraints to the model to restrict the instances (Figure 9, lines 2–4, between square brackets).

The participants noticed that user-defined types enable modelling on a level of abstraction that corresponds to domain objects, which prevents having to repeatedly model properties of an object such as a sheet separately (*DOMAIN-COVERAGE 2*).

The participants observed that the equality between input and output (Figure 9, line 13) is in terms of stacks, not in terms of the individual properties of stacks. Equality can thus be defined on a level of abstraction that corresponds to the objects modelled in CSX. This is in contrast to a low-level constraint modelling language, in which one would need to define equality with low-level constraints that compare each property of the stack individually (*DOMAIN-COVERAGE 3*).

Device. After having defined types that model uniform stacks, the participants started to actually model the device. This started with identifying the locations in the device where the stacks of sheets pass, typically just before and after the places in the process where modifications are made to the sheets.

The first iteration only contained an input and output location, both of type `Stack` (Figure 9). This is an oversimplification of the actual device: the model does include the physical limitations of the device, but it does not contain the different input trays, the stack cannot consist of different types of sheet (non-uniformity), and stitching and the possibility to orientate the sheets before and after stitching are not included.

The participants observed that the simplistic approach to modelling the device in this first iteration also led to a simple CSX model (*GO 5*). In the following iterations, as the level of detail in the models increased, additional locations were added by the participants such that more intermediate snapshots of the stacks could be considered in the model.

The participants observed that the stack of sheets at the input location of the device could be interpreted in two ways: they represent the total number of sheets that are in the input trays, or they represent the sheets in the input trays that will be used in the configuration for a single product. Alternatively, the configuration could also be used for multiple products in one job. In this model, the participants modelled the configuration space for single products. Therefore, the input of the device in the model represents the sheets of paper for a single job; there could be more paper in the physical tray.

The participants noticed that in the design process they did not use actions (see Section 2.3) yet to factor out common pieces of behavior, but modelled everything directly in a device. CSX supports actions for building a library of printing behavior that can be shared between many device models, but they were not used in the study (*GO 6*).

Inherent to the setup of the study, devices were modelled in separate CSX files. The incremental approach in the study has led to the insight that an import mechanism – which would allow re-use of, e.g., type definitions and actions between files – would be beneficial (*GO 7*).

Physical Limitations. The participants added physical limitations of the device in the first iteration (Figure 9, lines 10–11). The constraints in square brackets express the device's physical limitations with respect to the minimum and maximum size of sheets that it can handle. In this iteration, no maximum on the number of sheets was modelled. Note that the constant values that indicate the minimum and maximum width and height are integer values that represent a dimension for the precision chosen in this model. For example, the constraint `input.width <= 4500` indicates that the maximum width is 450mm.

Validation. Having a first simple model of the device, the participants wrote two tests to validate the physical constraints (Figure 9, lines 18–31). The first test checks that for a given input that is accepted within the physical constraints, the output contains the same stack. The second test checks that for an input that is too small, no configuration can be found (indicated with `fails`).

Also in the other iterations, the participants used tests to validate the behavior of the models. The tests evaluate after having changed the file, resulting in an interactive development experience. The domain expert observed that the modelling approach – in think aloud co-design sessions, with interactive validation using the tests – works well and stimulates experimentation. In particular, the domain expert noticed that the development and validation loop is quick (*GO 8*); updating the model and tests results into feedback within seconds.

The participants found it useful that CSX reports solutions found by solvers in terms of the CSX model instead of the generated constraint model. When inspecting a solution found by the solver, it is hard to map those to configurations of the device. Especially when lists are used, which are modelled with an array per property, it is difficult to understand which low-level values correspond to those of the CSX model. The participants observed that CSX is at a high level of abstraction when interpreting and presenting configurations (e.g., in tests): the configuration is reported in terms of the user-defined types and parameters, not in terms of low-level values (*GO 9*).

The participants observed that the modelling of objects in tests is still at a low level of abstraction (*DOMAIN-COVERAGE 4*). For example, to specify an input sheet object, one needs to specify each property of the sheet with individual constraints. Figure 16a depicts this: the test contains a `config` instance per property of the sheet that is relevant for the test. In this case, the thickness of the sheet is not relevant for the test, and thus omitted.

Non-uniform stacks of sheets. In iteration 2 (Figure 10), the participants aimed to increase the level of detail of the model by allowing stacks to be non-uniform. To do so, the participants refactored the model to use CSX's list construct for stacks of sheets (line 7). This enables to model non-uniform stacks, i.e., the sheets in the stack can have different properties (*DOMAIN-COVERAGE 5*). Since lists can have a variable size, the stacks can have a variable number of sheets. The participants noticed that a limitation of CSX is that although the upper bound is configurable, all lists get the same upper bound (*GO 10*).

Input trays. In iteration 3 (Figure 11), the participants first attempted to model input trays by combining the uniform stacks and non-uniform stacks. There are two input tray locations of type `UniformStack`. The case focuses on forming a non-uniform stack of sheets from the two input trays of uniform stacks of sheets. The idea behind the approach was as follows: the output stack should contain the sheets defined in tray 1 and those in tray 2, in any order. The counting is modelled by combining a map to a list of zeros and ones and then a sum. A count operator would help for expressing this (see commented part in Figure 11).

```
1  type Sheet {
2    width: int, [width > 0],
3    height: int, [height > 0]
4  }
5
6  type Stack {
7    sheets: list<Sheet>
8  }
9
10 device PaperInputModule {
11   location input: Stack
12
13   [input.sheets.forall {
14     sheet => sheet.width ≤ 4500 and sheet.height ≤ 3200}
15   ] // Max size
16   [input.sheets.forall {
17     sheet => sheet.width ≥ 1200 and sheet.height ≥ 1500}
18   ] // Min size
19
20   [output == input]
21
22   location output: Stack
23 }
```

Figure 10. The CSX model resulting from iteration 2 which includes the aspect of *non-uniform* stacks of sheets. The physical limitations of the device are expressed using a forall expression on the list of sheets in the input stack.

While modelling the input trays of the device, the participants noticed that the modelling of a non-uniform stack that contains sheets from two uniform stacks was challenging. In fact, the initial attempt was incorrect. In general, the handling of grouping and ordering of sheets and stacks remains difficult; the CSX user needs to incorporate several constraints that, e.g., enforce the total number of sheets to be correct (*DOMAIN-COVERAGE 6*).

In iteration 6, the participants re-modelled the tray assignment. This approach was also included in the final model (Figure 13). In this new approach, the participants included an enum with values for each sheet, and added a list that indicates the tray assignments for each sheet. By doing so, each sheet is actually from one of the trays – if a sheet gets tray A assigned, its value in the stack must be equal to the sheet defining the uniform stack in tray A. Additionally, the number of assignments per sheet are counted and compared to the number of sheets in the uniform stacks of the trays. This ensures that the total number of sheets add up. The participants observed that this was challenging to implement due to the constraint-based paradigm of CSX, as it failed on

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

type UniformStack {
  width: int, [width > 0],
  height: int, [height > 0],
  sheets: int, [sheets ≥ 0]
}

device PaperInputModule {
  location tray1: UniformStack
  location tray2: UniformStack

  [size(output) == tray1.sheets + tray2.sheets]

  [sum(output.map { sheet =>
    if (
      sheet.width  == tray1.width and
      sheet.height == tray1.height
    )
      1
    else
      0
  }) ≥ tray1.sheets]

  [sum(output.map { sheet =>
    if (
      sheet.width  == tray2.width and
      sheet.height == tray2.height
    )
      1
    else
      0
  }) ≥ tray2.sheets]

  // A count function could make above more expressive
  /*
  [count(input, { sheet =>
    sheet.width  == tray1.width and
    sheet.height == tray1.height
  } >= tray1.sheets)]

  [count(output, { sheet =>
    sheet.width  == tray2.width and
    sheet.height == tray2.height
  } >= tray2.sheets)]
  */

  location output: list<Sheet>
}
```

Figure 11. The CSX model resulting from iteration 3; the first attempt at modelling the input trays. In comments, it depicts how a count operator (which is not yet in CSX) could improve expressiveness. Note that this approach is incorrect for the case where the sheets in tray 1 and tray 2 are equal. Counterexample: both tray 1 and tray 2 contain one sheet with width 1, height 1, weight 1. The output stack could contain a sheet with width 1, height 1, weight 1 and a random second sheet, and still meet the constraints.

```
type UniformStack {
  sheet: Sheet,
  count: int, [count ≥ 0]
}

enum Tray { A B }

type Sheet {
  width: int,
  height: int,
  isPortrait = height ≥ width
}

type Stack {
  sheets: list<Sheet>,
  stitches: list<Stitch>
}

type Stitch {
  e: edge,
  direction: Direction
}

enum Direction { Upwards Downwards }
```

Figure 12. The type definitions accompanying the final CSX model (Figure 13).

the first attempt and required additional data structures and extensive testing to get right (*GO 11*).

Sheets must have equal height. In iteration 4, the participants included the constraint that all sheets must have the same height, which was also included in the final model (Figure 13, line 36). This captures a physical limitation of the device. The participants modelled this limitation using a forall that enforces all heights of the sheets to equal the height of the first sheet.

Edge Stitching. In iteration 5, the participants included the stitching capabilities of the device, which was also included in the final model (Figure 13). Again, for modelling the stitches, the first question that came to mind for the participants was which level of detail to include. The participants started with modelling stitches with an edge and a direction (Figure 12, lines 20–21). The e property has type edge, i.e., one of the geometrical contructs in this paper. A stitch can be applied in the upwards or downwards direction, which is modelled using an enum.

The model for the device could include the actual positions on which the stiches are applied on the sheets. In this model, we did not include the positions of stitches. The participants observed here that devices with similar features (in our case: stitching) can require different models (stitches with or without positions) (*GO 12*).

The device can apply multiple stitches to the stack of sheets, and therefore the participants used a list of stitches to model this. In addition to the usage of lists for modelling

```
device PaperInputModuleAndStitcher {
  location entryA: UniformStack [entryA.sheet.width ≤ entryA.sheet.height]
  location entryB: UniformStack [entryB.sheet.width ≤ entryB.sheet.height]

  parameter oA: orientation [oA == rot0 or oA == rot90]
  parameter oB: orientation [oB == rot0 or oB == rot90]

  [oA == rot0  implies entryA.sheet.width == trayA.sheet.width  and entryA.sheet.height == trayA.sheet.height]
  [oA == rot90 implies entryA.sheet.width == trayA.sheet.height and entryA.sheet.height == trayA.sheet.width ]
  [oB == rot0  implies entryB.sheet.width == trayB.sheet.width  and entryB.sheet.height == trayB.sheet.height]
  [oB == rot90 implies entryB.sheet.width == trayB.sheet.height and entryB.sheet.height == trayB.sheet.width ]

  location trayA: UniformStack
  location trayB: UniformStack

  [entryA.count == trayA.count] [entryB.count == trayB.count] [trayA.count ≥ trayB.count]

  location assignment : list<Tray> [size(assignment) == size(input)]

  location input: list<Sheet> [size(input) == trayA.count + trayB.count]

  [input.forall { sheet =>
    if (sheet == trayA.sheet) assignment[index] == A else (sheet == trayB.sheet and assignment[index] == B)
  }]

  var xA: list<int> var xB: list<int>

  [size(xA) == size(assignment) and assignment.forall { x => xA[ index ] == (if (x == A) 1 else 0)}] [sum(xA) == trayA.count]
  [size(xB) == size(assignment) and assignment.forall { x => xB[ index ] == (if (x == B) 1 else 0)}] [sum(xB) == trayB.count]

  [input.forall { sheet => sheet.width ≤ 4500 and sheet.height ≤ 3300}] // Max size
  [input.forall { sheet => sheet.width ≥ 1200 and sheet.height ≥ 1500}] // Min size

  [size(input) ≤ 50] // Max number of sheets that can be stitched

  [input.forall { sheet => sheet.height == first(input).height }]

  [gathered.sheets == reverse(input)] // Gathering a sequence of sheets will have the first sheet at the bottom of the stack

  location gathered: Stack [size(gathered.stitches) == 0] [output.sheets == gathered.sheets]

  [size(output.stitches) == 0 or size(output.stitches) == 2]
  [output.stitches.forall { stitch => stitch.e == right and stitch.direction == Upwards }]

  location output: Stack

  parameter o2: orientation [o2 == rot0 or o2 == rot90]
  [(o2 == rot0)  implies output.sheets.forall {
    sheet => sheet.width == reader.sheets[index].width and sheet.height == reader.sheets[index].height
  }]
  [(o2 == rot90) implies output.sheets.forall {
    sheet => sheet.width == reader.sheets[index].height and sheet.height == reader.sheets[index].width
  }]
  [reader.stitches.forall { stitch => output.stitches[index].e == orientate(stitch.e, o2) }]
  [output.stitches.forall { stitch => stitch.direction == reader.stitches[index].direction }]

  [size(output.stitches) == size(reader.stitches)] [size(output.sheets) == size(reader.sheets)]

  location reader: Stack
}
```

Figure 13. The final CSX model resulting from the co-design sessions. It integrates the key aspects of iteration 6 (properly modelling the input trays) and iteration 5 (edge stitching and orientations).

non-uniform stacks of sheets, the list construct in CSX is useful for coverage of a variable number of stitches (*DOMAIN-COVERAGE 7*).

The participants observed that both an edge and direction are geometrical constructs, but only the edge is provided first-class by CSX. Initially, the difference seems little. However, in iteration 5 the participants noticed that the support of edges in CSX is useful when applying orientations to the sheets with stitches. In this device, transformations of only 0 and 90 degrees are possible, which will not influence the direction of stitches. Therefore, this mode would not benefit from first-class support for directions such that they can also be transformed easily. However, in devices that can apply transformations that include flipping a stack of sheets with stitches, it would be useful if directions are supported first class with builtin transformations with orientations (*DOMAIN-COVERAGE 8*).

The participants noticed that geometrical constructs in CSX lift the level of abstraction on the modelling of geometric properties and transformations in CSX models. The geometrical constructs prevent the user from resorting to low-level linear algebra or handling many individual cases (*DOMAIN-COVERAGE 9*). Additionally, simple properties and transformations such as the modelling of an edge of a sheet actually becomes simple, e.g., when they need to be rotated. In CSX, orientations (including simple rotations) are part of the language and can be used to transform sheet sizes or edges. In our case, the participants used orientations to model the freedom of orientating the sheets before and after stitching, and we used it to model the edge of the stack on which the stitching occurs.

Orientations. In iteration 5, the participants modelled the aspect of orientations, which was also included in the final model (Figure 13). The device has the capability of orientating the stack of sheets before and after applying the stitches. This is useful because then a stack of sheets that is too large for being stitched in portrait orientation (lines 31–32) can be stitched in landscape orientation but still be presented in portrait orientation to the reader.

The `reader` location in the model represents a virtual location in which the operator has picked up the product and inspects it. In the printing domain it is common to include the reading in the model, because it enables us to reason about whether the end product conforms the intent of the operator. For example, on the reader location we could express the intent of a landscape orientated product with stitches on the left edge.

The capability of our device of orientating the stack of sheets has interaction with the geometrical constructs such as the edge on which a stitch is applied. The edge construct in CSX — including builtin transformations with orientations — makes it easy to express the transformation of an edge on a sheet (Figure 13, line 55). However, the participants noticed that such transformation on sizes are not built into CSX (Figure 13, lines 48–53). For example, when rotating a sheet by 90 degrees, the width and height are swapped. Although we can still express this in CSX with low-level modelling, CSX models would benefit from also having transformations of sizes expressed similar to edges (*DOMAIN-COVERAGE 10*).

Conclusions. We conclude our outcomes by summarizing the positive and negative observations regarding the domain coverage of CSX:

Positive observations:

- User-defined types enable modelling on a level of abstraction that corresponds to domain objects, which prevents having to repeatedly model properties of an object separately (*DOMAIN-COVERAGE 2*). Similarly, equality can be defined in terms of user-defined types and does not require comparing individual properties (*DOMAIN-COVERAGE 3*).
- The list construct in CSX contributes to covering the printing domain by enabling to properly model non-uniform stacks of sheets (*DOMAIN-COVERAGE 5*) or a variable number of stitches (*DOMAIN-COVERAGE 7*).

Negative observations:

- CSX does not cover precision and units: the modeller needs to choose a precision, and configurations that are found need to be interpreted under the chosen precision (*DOMAIN-COVERAGE 1*).
- Object terms, e.g., in tests, cannot be specified in terms of the domain, and need to be specified using low-level properties (*DOMAIN-COVERAGE 4*).
- Handling grouping and ordering of sheets and stacks is not specifically covered in CSX and thus remains cumbersome to model (*DOMAIN-COVERAGE 6*).
- The set of geometrical constructs in CSX is not complete and should be extended with directions (*DOMAIN-COVERAGE 8*) and sizes (*DOMAIN-COVERAGE 10*), because currently they require low-level modelling (*DOMAIN-COVERAGE 9*).

4.2 Configuration Accuracy

Study setup. To validate the accuracy of the CSX implementation, we test our implementation for *correctness* and *completeness*. For correctness, we test that the configurations that are found for a device correspond to the device's limitations. This ensures that there are no false positives. For completeness, we test that there are no configurations that are not found but that are possible in a device. This ensures that there are no false negatives.

To validate correctness and completeness, we test using artifical CSX models for which we manually determine

whether a configuration should or should not be found. In particular, we test the CSX language transformations and configuration space exploration. We approach testing systematically by covering all features of the language at least once in each language aspect (syntax, static semantics, desugaring, transformation to MiniZinc, integration with MiniZinc solvers, and interpreting MiniZinc solutions).

In addition to the systematic coverage of all language features in the tests, we add tests for specific cases of features that interact with each other. Because testing all feature interactions would be very time consuming, we test a subset of feature interactions. For example, the use of lists of edges in CSX involves a feature interaction between the specific way of translating lists to MiniZinc and that of translating edges to MiniZinc, and therefore is tested separately.

Results. Our testing has resulted in 232 handwritten unit and integration tests, which all pass and with that build confidence in the accuracy of the CSX implementation by covering all features at least once, and a subset of feature interactions, for correctness and completeness (*CORRECTNESS 1*).

Although the test suite covers a subset of the feature interactions of the CSX implementation, still there can be untested interactions between features that are not correctly handled by the implementation. While performing the coverage study, the participants exposed two bugs that were related to feature interactions. One of these bugs concerned the use of a list of a user-defined type in which a nested property was of an enum type. Although lists of user-defined types and lists of enums were tested, this particular case was not tested and required the handling of an edge case in the translation to Minizinc.

Although we made a best effort for testing accuracy also for feature interactions, based on the current test suite we cannot guarantee accuracy of all feature interactions (*CORRECTNESS 2*). In practice, specific interactions of features could lead to incorrect behavior or runtime failures.

Conclusions. We conclude the following on the correctness of CSX 2.0:

> A set of 232 unit tests generate confidence that all features, and a subset of feature interactions, in CSX contribute to configuration space exploration that is correct and complete (*CORRECTNESS 1*), but we cannot guarantee correctness and completeness for all feature interactions (*CORRECTNESS 2*).

4.3 Configuration Performance

Study setup. We consider the configuration space exploration performance to be practical when the complete pipeline of parsing, analyzing, and translating models into MiniZinc, finding a solution for the MiniZinc model, and translation back to CSX occurs in the order of seconds. This threshold is considered acceptable by Canon Production Printing's control software engineers for usage in interactive scenarios. In such scenarios, an operator interacts with a device by, e.g., describing an intent for a print job; getting feedback regarding the feasibility of this intent should not take longer than seconds in such cases.

Table 3. Scenarios of configuration space exploration for the final model of the coverage study (Figure 13) that we use for benchmarkking. The Di scenarios *derive* a configuration. The Oj scenarios find an *optimal* configuration by either minimizing or maximizing an objective. All scenarios use 10 as the upper bound on list sizes.

ID	Description
D1	Output landscape A3 with right edge stitch
D2	Output portrait A3 with stitches (which then need to be on the top edge)
D3	Portrait A3 with stitch on right edge (which is not possible)
O1	Derive smallest portrait size with right edge stitch
O2	Derive smallest landscape size with right edge stitch
O3	Derive smallest portrait size with top edge stitch
O4	Derive smallest landscape size with top edge stitch
O5	Derive largest portrait size with right edge stitch
O6	Derive largest landscape size with right edge stitch
O7	Derive largest portrait size with top edge stitch
O8	Derive largest landscape size with top edge stitch

Although the performance of constraint solving is hard to predict in general, we conduct experiments to get an idea of the typical response times for typical configuration scenarios at Canon Production Printing. In particular, we take the final model of the domain coverage study and we define realistic scenarios of configuration space exploration for it. Table 3 lists the scenarios that we consider, which includes three scenarios that *derive* a configuration (including one for which no configuration can be found) and eight scenarios that find an *optimal* solution. All scenarios use an upper bound on lists of 10 and all consider an output stack with five sheets.

We perform benchmarks to measure the performance for the different scenarios. Initial experiments and measurements have shown that the time spent on parsing, name binding, type checking, and translating solutions back to configurations is neglectable (<10ms), Therefore, in the benchmarks we only measure the time of translating a model and scenario to constraints and the actual solving time. We set a timeout of 10 seconds on the benchmarks (the upper bound of the order of seconds).

To get an impression of the impact of list sizes on performance, we repeatedly benchmark the first scenario for multiple list upper bounds. For that, we alter scenario D1 such that the output stack size that is considered is half of the upper bound on lists. This ensures that the lists in the

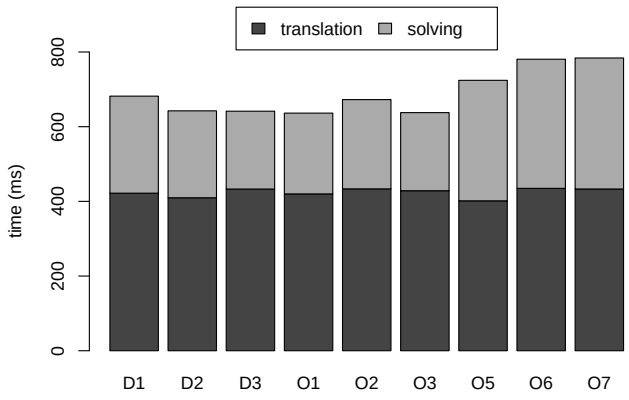


Figure 14. Benchmarking results for the Gecode solver for the scenarios from Figure 3. The bars show the translation and solving time separately. Times are reported in milliseconds. The tests D4 and D8 timed out and therefore are not included in the figure.

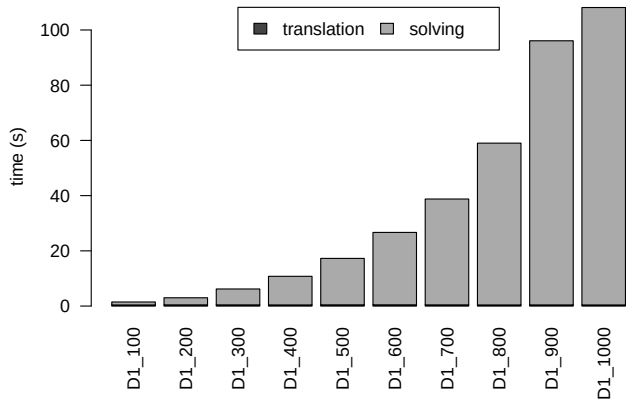


Figure 15. Benchmarking results for the Gecode solver for scenario D1 with list upper bounds varying from 100 to 1000 in which the output stack size is half of the upper bound on lists.

MiniZinc solution both have relevant values (that are considered in the configuration) and framed values (which are ignored). For example, for the test with an upper bound of list sizes of 300, the scenario considers and output stack of 150 sheets. For these benchmarks, we do not set a timeout.

We use the JMH framework[1] to implement the benchmarks, which is a framework for benchmarks in Java. Spoofax offers a core library that allowed us the integrate the relevant components of CSX in the benchmark such that we can measure the translation time and solving time separately. We executed the benchmarks on a laptop with a quad-core processor with a base frequency of 3.1GHz and 16GB RAM, running macOS 12.4 and using Java version 1.8.0_275. Furthermore, we used version 2.6.4 of MiniZinc with two common solvers [17]: Gecode[2] and ORTools[3]. For each scenario, we first run 10 warmup iterations, then run 10 measurement iterations, and we report the average of the measurement iterations. We only report the results of the best performing solvers, based on the least timeouts.

Results. The Gecode solver completed the benchmarks with the least timeouts, and therefore we only report the Gecode benchmark results. Figure 14 depicts the benchmarking results for the Gecode solver for all scenarios. The results show that most of the scenarios succeed within one second, and thus stay within the order of seconds time limit (*PERFORMANCE 1*). Two of the optimization scenarios (D4 and D8) — although they seem comparable to the other optimization scenarios — timed out (*PERFORMANCE 2*). For all scenarios, the translation times are higher than the solving times. The ORTools solver on average performed better on the derivation scenarios, but it timed out on all optimization scenarios.

Figure 15 depicts the benchmarking results for increasing list upper bounds on scenario D1. The results indicate that increasing the list upper bounds negatively impacts performance (*PERFORMANCE 3*). This is expected, as increasing the list upper bound increases the solution space in which solvers need to find a solution. It is unclear yet for which cases in practice this could become problematic.

Conclusions. We conclude the following on the performance of CSX 2.0:

> For several scenarios of configuration space exploration on a model of a device at Canon Production Printing, the performance is in the order of seconds and thus acceptable for interactive configuration space exploration (*PERFORMANCE 1*). However, performance is unpredictable, because for seemingly similar scenarios the solving can also time out (*PERFORMANCE 2*). Increasing the upper bound on lists sizes increases the solution space and negatively impacts performance (*PERFORMANCE 3*).

4.4 Relevance

Study Setup. To evaluate the relevance of CSX 2.0, we gather anecdotal evidence by interviewing the domain expert and by considering general observations from the coverage study (Section 4.1). In particular, for the relevance of CSX 2.0 for developing control software for printing systems, we consider sufficiency and necessity:

[1] https://openjdk.java.net/projects/code-tools/jmh/
[2] https://www.gecode.org
[3] https://developers.google.com/optimization

Sufficiency (CSX vs. pre-CSX). Is it sufficient to use CSX 2.0 to realize automatic configuration space exploration, resulting into an improvement over the pre-CSX situation?

Necessity (CSX vs. MiniZinc). Is it necessary to use CSX 2.0 instead of directly modelling printing systems in a generic constraint modelling language such as MiniZinc?

Sufficiency. To compare CSX with the pre-CSX situation, we look at the features that CSX introduces and the potential impact of CSX on the development process.

The domain expert mentions that the biggest strength of CSX is that based on a model, a solution space is derived automatically and (optimal) configurations can be found automatically (*RELEVANCE 1*). The domain expert characterizes this as levelling up automation. This was the main objective when starting the development of CSX. In that respect, CSX is an improvement over the pre-CSX situation.

In the pre-CSX situation, device operators do trial and error to find a configuration, and are minimally assisted by the control software. CSX's ability to realize and automate configuration space exploration is the biggest advantage over the pre-CSX situation (*RELEVANCE 2*). For example, taking the example of the edge stitching case, deriving automatically what the maximum end size is that can be stitched left is something that is possible with CSX but which was not possible with pre-CSX.

The domain expert reports that a key change in CSX with respect to pre-CSX is the language's declarative nature. With CSX, modelling the printing system only concerns thinking about the characteristics of devices, and not about *how* to compute or find configurations for the devices. Given a CSX model, the configuration space exploration becomes an independent concern that can be fully automated (*RELEVANCE 3*).

In the pre-CSX situation, control software engineers develop heuristics to automatically find (partial) configurations in order to improve usability of the devices. Typically, the heuristics cover many individual cases by branching on particular input and parameter values, resulting in large decision tables. Those decision tables typically do not cover the full configuration space and are not composable. Therefore, the heuristics are for single devices, hindering reusability and maintainability. With CSX, no algorithms need to be developed for realizing the configuration space exploration, limiting the repeating work when modelling new devices (*RELEVANCE 4*).

The domain expert mentions that the development time of control software for printing systems could be greatly reduced if CSX would be deployed in practice (*RELEVANCE 5*). The domain expert estimates the currently required development time required for integrating a device similar to the one in our coverage study to be four to eight man weeks. This development time can be reduced because repeating work for new devices is decreased with CSX. The interactive testing facilities of CSX allow modellers to validate parts of their models already in the IDE (*GO 8*), decreasing the time-costly dependency on physical hardware for validation.

Without claiming to make a fair comparison, we have asked the domain expert to make an estimation of lines of C# code in the pre-CSX situation that would cover the same concerns for a similar case as in our coverage study. The estimation was in the order of thousands lines of code. Our CSX model consists of less than hundred lines of code. Thereby, the estimation indicates that the lines of code involved in modelling a device can be reduced by an order of magnitude.

Necessity. Although CSX could be beneficial with respect to the pre-CSX situation, the question remains whether it is worth it to develop a new language instead of using a generic constraint modelling language such as MiniZinc.

The domain expert reports that using a generic constraint modelling language for modelling printing systems could already give benefits. One could write a constraint model that represents a configuration space, and have solvers find solutions which correspond to configurations. However, a problem with this approach is that domain-specific aspects in print systems need to be modelled repeatedly in low-level constraints, as they are not available in the language. MiniZinc does support constructs that facilitate reuse such as functions. Recently, MiniZinc also added support for record types. Still, MiniZinc lacks domain-specific support for, e.g., device and action modelling. The domain expert thinks that support for domain-specific aspects in the modelling language is required to make modelling using the language feasible in practice (*RELEVANCE 6*).

The domain expert also mentions the level of abstraction as a key characteristic that makes CSX more realistic to use in practice than MiniZinc (*RELEVANCE 7*). The domain expert reports that CSX is capable of abstracting over the complexity of low-level constraint modelling, by offering high-level language constructs.

In addition to the domain-specifity and level of abstraction of CSX, the domain expert mentions the benefits of the CSX IDE (*RELEVANCE 8*). For example, the CSX IDE provides inhabitance checking and test feedback. These features are interactive which speeds up the development process. Also, the configurations found in tests are reported while hovering over a test with your mouse, making it accessible to inspect configurations.

Conclusions. We conclude the following on the relevance of CSX 2.0:

- CSX is relevant because it realizes configuration space exploration (*RELEVANCE 2*) that is automatic

(*RELEVANCE 1*) by only modelling device characteristics (*RELEVANCE 3*) and without requiring repeating development of algorithms for new devices (*RELEVANCE 4*).
- CSX is relevant because it has the potential of increasing control software development productivity by greatly reducing development and validation time (*RELEVANCE 5*).
- CSX is relevant because, in contrast to a generic constraint modelling such as MiniZinc, the language includes constructs specific to the printing domain (*RELEVANCE 6*) which are on a higher level of abstraction (*RELEVANCE 7*), accompanied with an IDE with useful features such as inhabitance checks and interactive testing (*RELEVANCE 8*).

```
type Sheet {
  width: int,
  heigth: int,
  thickness: int
}
device MyDevice {
  location in: Sheet
  ...
}
test device MyDevice
config in.width = 210
config in.height = 297 {
  ...
}
```

(a) CSX 2.0: individually specified properties.

```
type Sheet {
  width: int,
  heigth: int,
  thickness: int
}
device MyDevice {
  location in: Sheet
    ...
}
test device MyDevice
config in = Sheet(210, 297, _) {
    ...

}
```

(b) Hypothetical CSX 3.0: an object term with a wildcard.

Figure 16. The partial model of a sheet object in a test in CSX 2.0 and hypothetical CSX 3.0.

5 Discussion

In this section, we discuss CSX's language design, the implications of using constraint-based programming, and CSX's application in practice more broadly. If relevant, we refer to observations of the coverage study.

5.1 Language Design

We discuss the implications of CSX's language design decisions and we discuss ideas for improving the language design.

User-Defined Types. When modelling with CSX, the level of detail that is included is an important design question. For example, when modelling the stitches that get stitched in a stack of sheets, is it necessary to only model the existence and number of stitches, or should the exact locations of the stitches also be included? For some stitching devices, only the number of stitches needs to be indicated and the device will position them automatically. For other stitching devices, the exact position of the stitches needs to be configured. Because CSX offers user-defined types to model objects, the modeller retains flexibility in choosing what to include in the object representations.

We consider that user-defined types should be used to model the objects of printing and finishing devices as the most import language design decision of CSX. It influences the modelling process in such a way that modelling starts with types (*GO 2*) and that the modeller remains flexible by iteratively including more detail in types (*GO 3*). We think that this characteristic of CSX is essential in making sure that a simplistic approach to modelling a device also leads to a simple model (*GO 5*), not polluted by irrelevant details.

Alternatively to user-defined types, CSX could offer built-in constructs for its objects (sheets, stacks, stitches, etc.). This would make the language more domain-specific, but also less flexible, which is a typical tradeoff in language design. Already for a simple device such as a stitcher, it would not be obvious to use a single type definition for a stitch (as it could be necessary with and without position information). Possibly, CSX could offer both built-in type and user-defined types to be more domain-specific but also maintain the flexibility.

From our evaluation, we cannot conclude whether the freedom in type definitions is also effective when covering a larger and more diverse range of printing systems. Although user-defined types give freedom in how printing objects can be modelled, possibly specific for a particular device, the anticipated reusability of type definitions could be hindered when a wider range of devices are modelled.

Units & Precision. CSX 2.0 does not support units in the language. A modeller is restricted to using integers and has to choose a precision, which also requires manual interpretation of configurations for that precision (*DOMAIN-COVERAGE 1*). We could overcome the need of this manual interpretation by introducing units in the type system of CSX, such that the values and their types reflect actual measures. Potentially, this could be used to extend CSX such that a user can experiment with varying precisions without having to update the complete model.

Object Constructors. In Figure 16b we depict how the modelling of objects could be improved in a next version of CSX. By introducing object constructors, the test object can be specified in terms of user-defined types. If a property of the object is not relevant, it can be ignored by using a wildcard, which means the property could get any value. We expect extending CSX with support for object constructors with wildcards to be relatively straightforward.

Sizes. In Figure 17, we compare modelling the transformation of sheet sizes in CSX 2.0 (similar as in our evaluation case) with an alternative approach in a hypothetical CSX 3.0.

```
1  enum Color { White Red }
2  type Sheet {
3    width: int,
4    height: int,
5    color: Color
6  }
7  device Rotator {
8    location input: list<Sheet>
9
10   parameter o : orientation [o == rot0 or o == rot90]
11
12   [o == rot0  implies input.forall { sheet => sheet.width == output[index].width  and sheet.height == output[index].height }]
13   [o == rot90 implies input.forall { sheet => sheet.width == output[index].height and sheet.height == output[index].width  }]
14
15   [input.forall { sheet => sheet.color == output[index].color }]
16
17   location output: list<Sheet>
18 }
```

(a) CSX 2.0: low-level modelling of orientation the sizes of sheets. More cases such as on line 12 and 13 would be needed if the device would support more orientations than only 0 and 90 degrees. The color of sheets which is not changed by the rotation, are mapped to the output (line 16).

```
1  enum Color { White Red }
2  type Sheet {
3    size: size,
4    color: Color
5  }
6  device Rotator {
7    location input: list<Sheet>
8
9    parameter o : orientation
10
11   [output == input.map { sheet => Sheet(orientate(sheet.size, o), sheet.color) }]
12
13   location output: list<Sheet>
14 }
```

(b) Hypothetical CSX 3.0 with two new features. First, by adding first-class support for sizes, the `orientate` function can also be used for transforming sizes, removing the need of manually writing out cases for each orientation. Second, by adding object constructors, a single map operation can be used to express an effect on a list where some properties do change (i.e., size) and some not (i.e., the color of sheets.)

Figure 17. Modelling the transformation of sheet sizes in CSX 2.0 (low-level) and in hypothetical CSX 3.0 (high-level).

By extending the set of geometrical constructs in CSX with sizes, size transformations can be modelled without having to model independent cases. Additionally, by using object constructors, a map operator can express a change over a list of items by conveniently modelling which properties do and which do not change.

Lists. The list construct in CSX contributes to the coverage of CSX for the printing domain (*DOMAIN-COVERAGE 5*), as non-uniform stacks allow to include more detail in the model. Realizing a variably sized non-uniform stack of sheets in principle would be possible without the list construct, but it is cumbersome. Figure 18 demonstrates this.

Lists allow to incorporate properties such as paper type, color, and width in the sheet model and accept stacks of sheets with variation in those properties. Additionally, aspects such as a variable number of stitches can be modelled properly with a list.

Note that stacks do not necessarily have to be modelled in a non-uniform way. If it is clear for a model that a particular stack is uniform, it could be better to model it as such. This is a more efficient representation, as it requires the modeller to only needing to model the width and the height of the stack once, instead of for each sheet in the stack separately. Also, if a uniform stack would be split up in multiple stacks, the new stacks could still be considered as uniform stacks.

CSX 2.0 supports a single point of configuration for list upper bounds (*GO 10*). If it is known that a list will have a small maximum size, e.g., for a device that can only stitch 6 stitches maximum, it would be a better and more efficient model of the solution space if the instance of a specific list

```
type Sheet {
  width: int, heigth: int
}
type Stack {
  sheet1: Sheet,
  sheet2: Sheet,
  ...
  sheet5: Sheet,
  size: int,
  [0 ≤ size and size ≤ 5]
}
```

(a) CSX 1.0: a sheet instance for each possible sheet in the stack, in which the variable `size` indicates which sheets should actually be considered in the tack.

```
type Sheet {
  width: int, heigth: int
}
type Stack {
  sheets: list<Sheet>
}
```

(b) CSX 2.0: using lists to represent a non-uniform stack of sheets.

Figure 18. Example type definitions for modelling a non-uniform stack of sheets in CSX 1.0 (with a workaround) and in CSX 2.0 (using the `list` construct).

could get its own upper bound. We expect extending CSX with upper bounds for lists that are configurable per list instance to be relatively straightforward.

Reuse. In the coverage study, the participants did not yet make use of the actions language construct (Section 2.3) (*GO 6*). We expect the reason for this to be that actions are useful for factoring out common pieces of behavior (for which they were intended), but that it only becomes useful when a wider range of devices are modelled. To get a better understanding of the usefulness of actions in capturing reusable parts of printing behavior, we need a study on more devices. For a library of actions to be useful in practice, we think it is necessary that CSX also supports importing (*GO 7*). This would enable that the library of actions can be defined separately and types and actions from the library can be imported in specific device models.

Challenging Patterns. Although CSX offers various constructs that ease the modelling of printing systems, we encountered several patterns that remained difficult to model. Two examples are the input trays and orientation of a stack.

Modelling the input trays in the coverage study was considered challenging by the study participants (*GO 11*). In Figure 13, the code for modelling input trays is duplicate for trays a and b. Although this part could be factored out in an action to become reusable, it still would require redundant modelling for cases with more than two trays.

In the coverage study, the device was limited to rotating the stacks by 0 or 90 degrees. When the set of possible orientations would be increased, the number of orientations cases that need to be handled such as in Figure 13 (lines 48–53) grows. Partially, this manual handling of orientations could be resolved by supporting sizes (*DOMAIN-COVERAGE 10*) and object constructors (see Figure 17). However, when the orientations with flips are allowed, this still not suffices. When a stack is flipped, the order of the sheets also becomes reversed.

To improve support for these patterns, CSX could be extended by adding domain-specific constructs or generic expressive power. CSX could be extended with additional abstraction mechanisms that support modelling common patterns such as input trays or stack orientation. Alternatively, CSX could be extended with generic abstraction mechanisms that facilitate reuse of code.

5.2 Constraint-Based Programming

Paradigm Shift. Although CSX is on a high level of abstraction, it still is a constraint-based language. Constraint-based programming is typically not in the skillset of an average control software engineer. Our domain expert, who is an experienced object-oriented and functional programmer, but who had no experience with constraint-based programming before we started working on CSX, experienced a steep learning curve when starting with constraint programming in either CSX or MiniZinc.

The domain expert reports that seemingly simple aspects require unintuitive modelling in CSX. An example of this is the modelling of the tray assignment in the case of the coverage study. Possibly, CSX could be extended with constructs that abstract over unintuitive but common modelling patterns. Still, CSX would remain a constraint-based language, which involves a paradigm not familiar to programmers working with object-oriented or functional programming languages, and we consider this as a critical risk for its applicability in practice.

Another characteristic of constraint-based programming in CSX is that also properties that do not change between locations have to be defined as equal in both locations. This is counterintuitive for a programmer used to functional programming, as you do not need to specify things that do not change in functional programming. In constraint-based programming, we could see the need for specification of things that do not change as modelling overhead. Possibly, CSX could be extended with constructs that easy the modelling of non-changing properties.

The domain expert reports that interactive tests and the possibility to easily inspect configurations for debugging helps in overcoming unintuitive modelling tasks. In the case of an unexpectedly failing test, the user can easily inspect the found configuration under which the test fails. Also, if the test contains multiple assertions, the IDE indicates which of the assertions fails for the found configuration.

Level of Detail and Solving Performance. In theory, one could go as far as modelling a sheet of paper as a set of atoms. In practice, that would not be feasible with respect to solving performance, and it also does not have practical utility. In our work, the question remains what the actual needed level of

detail in a model needs to be. In general, modelling with CSX involves a tradeoff between including more detail on the one hand and improving performance on the other hand. Based on our current experiences, we cannot yet conclude if CSX would have performance that is good enough for integration in UIs for all printing devices.

Currently, we have only evaluated CSX with two common solvers using the default settings and default search strategy. Possibly, specific settings or search strategies can improve solving performance for MiniZinc models that correspond to CSX models. Also, our performance evaluation shows that for the reported cases, the translation time was higher than the solving time. Since we have not performed any performance engineering at all on the transformation implementations, possibly the translation times can be improved as well.

Although in our evaluation we have focussed on performance for interactive usage scenarios which have a strong demand on performance, longer solving times could be permitted in other scenarios. For example, once it is confirmed that an operator's intent can be realized, it would be acceptable to wait longer for finding an optimal configuration for the intent that, e.g., minimizes paper waste. In particular, a longer waiting time is acceptable for large volume jobs, e.g., printing hundreds of books. In general, there is a balance between solving time and job volume and execution time; the larger the job, the more solving time can be permitted up front.

It could occur that for a realistic model the solving performance is not sufficient for usage in interactive scenarios. In such cases, the model would possibly still be useful for validation of devices, as orders of magnitude slower performance are still acceptable if it can be used to derive edge cases in the configuration space for physical validation of the device. Alternatively, the level of detail in the model could be reduced such that it can be used for coarse-grained configuration space exploration.

CSX 2.0 currently only supports integers for modelling dimensions, not floating point or real numbers. Although MiniZinc does support solvers that support floating point numbers, early experiments indicated that performance quickly drops when using them. Therefore, we have not further explored the use of floating point numbers for modelling in CSX.

Currently, we have used SMT constraint solvers for all our experiments. For many devices, general solvers were necessary because the configuration spaces correspond to problem spaces that include a mix of linear, satisfiability, and logical constraints. In practice, we could encounter printing devices for which the configuration space corresponds to a more restricted set of problems, e.g., linear problems. In such cases, we could employ more specific solvers, e.g., linear solvers, to improve solving performance for these specific devices.

Browsing Configurations. CSX is currently limited to presenting a single configuration, although multiple configurations could be possible for a scenario. Potentially, it could be useful to visualize the space of configurations that are found such that an operator can get insight in what flexibility in configuration remains for a scenario.

Although CSX does support optimizing for a given objective, in practice an operator might be interested in choosing between *multiple* objectives. Possibly, existing multi-objective optimization approaches could be ported to CSX and a user interface to assist operators in choosing between multiple objectives, e.g., to answer questions such as "If I can afford to waste some more paper, how much productivity gain does that offer me?".

Traceability. The current version of CSX only reports a single configuration for a requested (partial) configuration or job specification, or it reports that no configuration is possible for a job. If no configuration can be found, there is no further indication of *why* no configuration can be found. In practice, this would hinder the usability of the system for operators. Possibly, existing approaches for identifying minimal unsatisfiable sets [13] could help in tackling this. Then, characterizations of minimal unsatisfiable sets should be mapped from the constraint level back to the CSX level to make them understandable for operators.

5.3 Application in Practice

We discuss aspects related to CSX's applicability in practice at Canon Production Printing.

Integration with Control Software and UI. CSX currently solves the problem of modelling devices and realizing automated configuration space exploration, but requires realization of more of the components in the architecture of Figure 6 for application in practice. Realizing these components requires a substantial investment, but the potential software engineering productivity gains and added functionalities can compensate that investment. The two most important components that currently are missing are the integration with a user interface and code generation for instructing low-level embedded software.

Although we have realized configuration space exploration for realistic cases and useful scenarios, still, the scenarios need to be described in a rather low level format (in CSX itself). For CSX to be applicable in devices, there should be an integration with a user interface targeted at end users (print system operators). Such an interface is typically visual in which the user can specify a partial configuration and get feedback on it, rather then describing it in text in an IDE. To use CSX for finding validation scenarios, the existing IDE can already be used by control software engineers.

The aim of CSX is to realize configuration space exploration that is automatic and to have an effective and scalable method for integrating a large range of finishing devices. The

integration of a device comprises more than just the modelling of the configuration space. Infrastructure is needed to — for a given configuration — instruct low-level embedded software components to operate under a configuration. The pre-CSX software already tackles this concern and thus CSX can become a layer on top of pre-CSX, generating the low-level control software components.

If CSX would be integrated in production control software, this adds dependencies on external software components. Spoofax would not be required to include in the control software, as Spoofax can generate language artifacts for compiling CSX models and integrate those with solvers, and only those artifacts need to be added. A solver does need to be integrated in the control software, as it is required for configuration space exploration.

Learning Curve. To successfully apply CSX at Canon Production Printing, the company would need to train developers to work with CSX. In particular, control software engineers need to be introduced to constraint-based programming and then to CSX in particular.

Language Engineering. Using a DSL to develop software in a company introduces a dependency on language engineering. In our work, the use of a language workbench has done much of the "heavy lifting"; Spoofax provided and automated a large part of the language infrastructure for free, by generating parsers, compilers, and an IDE from language specifications.

Still, experience with language engineering — and in our case with Spoofax in particular — is required to understand, maintain, or evolve the language implementation. Since there are few programmers with such experience available, and because there is a significant learning curve in language engineering, there is a risk of using a DSL without having the resources to maintain the language. However, although the introduction of language engineering in control software development adds external dependencies and new skills to be learned, it has the potential to outweigh those drawbacks with the productivity gains and complexity reductions that the approach realizes.

Besides the dependency on language engineering as a skill, our implementation of CSX in Spoofax also imposes a dependency on the Spoofax tool. Although we found that Spoofax was effective for the implementation of CSX, we think other state of the art language workbenches [6, 8, 12] could be used as well. If another tool for language development becomes preferred, the CSX language implementation could be ported. CSX has textual syntax, which eases the migration to another tool as the grammar can be ported, and existing CSX models can be maintained. With visual syntax or projectional editing this migration could be less straightforward.

The CSX implementation uses MiniZinc as the target language for expressing constraint models and interfacing with constraint solvers. Because MiniZinc is a solver-independent language and supports multiple solvers, there is no dependency on one solver in particular. Although we found MiniZinc an effective target language for generating constraint models, we think that CSX could also be realized with alternative languages for expressing constraint models and interfacing with solvers.

Domain Specificity. Although we have designed CSX specifically for the printing domain, the language only has a few features that are specific to printing. We could see CSX as consisting of three layers in which the bottom layer contains standard constraint programming and only the top layer makes it specific to printing. For example, in the top layer, CSX supports a restricted set of eight orientations which are specific to printing and sheets. In the middle layer, CSX's device, action, and location concepts make the language potentially applicable to a broader field of flexible manufacturing systems, i.e., manufacturing systems that have no predefined set of possible products to manufacture. We can characterize such systems as follows. First, the manufacturing systems do not just assemble input materials, but can also modify the materials. Second, the modifications are not fixed but are configurable and thus span a configuration space. Especially if it is challenging to find valid or optimal configurations, then CSX could be useful. Because CSX allows to define types in the language for modelling materials, it could cover manufacturing systems that handle other materials than paper.

5.4 Lessons Learned

We list our most important lessons learned on applying a constraint-based DSL in an industrial context:

1. The Spoofax language workbench and the MiniZinc constraint modelling language (and compatible solvers) took care of much of the "heavy lifting" in realizing CSX. This enabled us to tackle complexity and improve functionality in software engineering for a complex domain by allowing us to mostly focus on the domain and language design.
2. A systematic approach to DSL evaluation is useful for communicating about a DSL in an industrial context. Concrete evaluation criteria for the use of a DSL help in the discussion to explain to people who have no experience with DSLs to understand what is required for a DSL to be applied in practice. Finally, the evaluation criteria guide decision making regarding adoption of the technique.
3. Starting to use a DSL in practice has a big impact on the software engineering process with dependencies on external tooling and having language engineering resources available for both language development and language maintenance. Therefore, the benefits

of adopting a DSL need to be large to outweigh the corresponding investment.
4. The conceptual power of CSX is amplified by its IDE. The CSX IDE gives helpful insight in the behavior of models by featuring interactive validation of tests and debugging through inspection of configurations. This helped us to rapidly prototype and try out new language designs, leading to an iterative language development process.
5. It is a crucial language design decision to have types being defined in a language itself — instead of embedding a fixed set of domain objects in the language — which enables flexibility in modelling by iteratively including more detail in models.
6. A high level of abstraction and domain-specific constructs such as in CSX are necessary to make constraint-based modelling accessible. Still, 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.

5.5 Threats to Validity

The nature of our study raises threats to validity, which we discuss below.

External Validity. We have presented an experience report that focusses on a particular industrial context, and therefore we do not claim that our findings are generalisable. Still, we think the outcomes of our work can be useful to others working in an industrial context where a domain-specific interface to constraint solving is useful. Ultimately, we need to further apply CSX on a wider range of printing systems and with more engineers and domain experts to get a better understanding of the effectiveness and scalability of the method.

We have described the protocol of our coverage study to promote replicability. CSX 2.0's source, tests and benchmarks cannot be published due to confidentiality reasons, hindering reproducibility of tests and the benchmark results. In order to reproduce the results, others would need to manually create a CSX implementation and set up similar studies.

Internal Validity. Two authors were also the participants in the coverage study, which raises a concern with regard to *confirmation bias*, or the tendency to search for evidence supporting prior beliefs. We have tried to mitigate the risk of confirmation bias, by openly communicating about each step of the evaluation, and about each observation made, with the other authors of the paper.

Construct Validity. The accuracy of the configuration space exploration that we have studied in this paper is dependent on the CSX language, IDE implementation, and the CSX models. We have countered this threat to construct validity by testing the CSX implementation and by writing tests for the CSX models that we have written. The accuracy study relies on the tests itself, which could test for incorrect expectations. We have countered this threat by carefully determining the expectations for all tests manually.

The measurements of benchmarks could be influenced by many factors. We have countered this threat to construct validity by running the benchmarks on a computer which has most other applications disabled and is disconnected from network access. The benchmark's first 10 runs were considered as warmup iterations. We considered the subsequent 10 iterations for measurement. We report the average of these 10 measurements.

6 Related Work

We describe related work in which high-level modelling languages interface with constraint solvers in the backend. We focus on more general constraint solving approaches as our objective made us select SMT constraint solvers for CSX and because our practical experience showed that applying CSX involves models with various types of constraints (linear, logical, satisfiability). Whereas other work focusses on evaluating the tools used to create DSLs [12, 20], we focus on evaluating the DSL itself.

The work of Keshishzadeh et al. applies constraint solving in the backend of a DSL for the domain of medical imaging equipment [11]. In particular, they use constraint solving to validate domain-specific properties for realizing collision prevention in the equipment. If such properties are violated, the causes of violations can be traced through delta debugging and reported back on the model-level.

KernelF by Voelter et al. is a reusable functional language for the modular development of DSLs [19]. KernelF features advanced error checking and verification based on constraint solving with the Z3 solver. In a case study on payroll calculations [21], these techniques are applied to statically check completeness and overlap of domain-specific switch-like expressions. These forms of static analysis are similar to the interactive analysis of CSX.

Although in this paper we have focussed on CSX as a method to realize automatic configuration space exporation, the language also has the potential to cope with the large variety of finishers. The use of constraint solving is common in product line engineering, and, e.g., also used in feature models of printing systems [15], but constraint solving in that context has a different utility than in CSX. Feature models can be used to model systems as compatible compositions of features or components, and constraint solving can be used to find or check feature compositions. CSX, in contrast, is used to find configurations at run time for a particular device.

De Roo et al. [4] present an architectural framework for realizing multi-objective optimization for embedded control software. Additionally, they introduce a toolchain that consists of visual editors, analysis tools, code generators, and weavers. The approach is based on domain-specific models from which optimization code is generated automatically. Both CSX and their work use constraint models for realizing control software and support solving for optimization objectives. The authors evaluate their work in the context of the industrial printing domain as well. Roo's DSL is targeted at a different sub-domain of printing software, namely embedded online control. Our domain represents the configuration spaces of a product family of hardware devices, and the configuration control software that can be derived from it. Our work on CSX is different in the sense that it is used before the execution of print jobs (offline) to derive configurations, whereas de Roo et al. focus on optimization in embedded control software that runs during the execution of print jobs (online), imposing different requirements. Finally, the aim of CSX is to involve domain experts such as mechanical engineers in the modelling process.

Constraint solving is also used in model checking and relational model finders. For example, Alloy [9] is a high-level specification language that features finite model finding to check formal specifications. Alloy uses KodKod [18], which is a relational model finder on problems expressed in first order logic, relational algebra, and transitive closures. KodKod differs from CSX in several ways. In KodKod the nature of models is relational, where CSX considers fixed manufacturing paths and models the objects and parameters in such paths. KodKod does not support reasoning over data or optimization, where CSX does support optimization.

Stoel et al. extend relational modeling finding with first-class data attributes and optimization in AlleAlle [16]. Similar to CSX, AlleAlle includes data into problem models and uses SMT constraint solving for modeling finding. CSX and AlleAlle differ in the sense that AlleAlle is an intermediate language that targets relational problems, while CSX is a DSL specific to the printing domain and without first-class support for relations. AlleAlle and CSX both lack an approach for mapping reasons for unsatisfiability that are found on the constraint level back to the model level.

Muli [3] integrates constraint solving with the object oriented programming paradigm by extending the Java programming language. Muli adds support for symbolic values to Java, which translate to constraint variables in the runtime. Muli features a runtime that integrates constraint solvers in a Java virtual machine. In contrast to CSX, Muli is a general purpose programming language, and it does not support lists or optimization.

Although our work on CSX contains parts that are similar to other high-level modelling approaches with constraint solving backends, the distinctiveness of our work is that we extensively worked out a full stack implementation for a specific domain and evaluated it thoroughly in an industrial context.

7 Conclusions

We have presented CSX 2.0, an extension of the CSX language and environment for the development of control software for digital printing systems. We extended the language's coverage by adding support for lists and high-level support for geometrical constructs. To bring the constraint-based language closer to the functional programming paradigm, we added functional-style operators that get translated automatically into predicate-style counterparts. If this translation requires intermediate variables, those variables are automatically added.

We have qualitatively evaluated CSX by having the developer of CSX and a domain expert model a realistic device in think aloud co-design sessions. We find that CSX is suitable for covering a large part of the printing systems domain, although coverage for some parts can still be improved. A major hurdle for adoption of CSX is its declarative paradigm; it is hard — even for experienced developers — to switch from more traditional programming paradigms to the declarative programming style. Quantitative evaluation using bencharks confirms that CSX has reasonable runtime performance for realistic scenarios.

7.1 Future work.

We plan to apply CSX on a wider range of devices to further evaluate its effectiveness and scalability. To improve solving performance, we intend to assist solvers in their search by providing domain-specific information. Ultimately, we envision CSX as a language that could also be used by domain experts such as mechanical engineers, in which, e.g., usability of the language and maintainability of the models would be of vital importance; we consider evaluation of such dimensions as future work.

Acknowledgments

This research was partially supported by a grant from the Top Consortia for Knowledge and Innovation (TKIs) of the Dutch Ministry of Economic Affairs and by Canon Production Printing. This work is related to the European patent application EP3855304 A1 which is published on 28 July 2021.

This study was started under the guidance of Eelco Visser, who passed away on April 5th, 2022. The authors decided to posthumously acknowledge his contributions to this work by making him co-author.

References

[1] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2021. Satisfiability Modulo Theories. In *Handbook of Satisfiability - Second Edition*, Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). Frontiers in Artificial Intelligence and Applications, Vol. 336. IOS Press, 1267–1329. https://doi.org/10.3233/FAIA201017

[2] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming* 72, 1-2 (2008), 52–70. https://doi.org/10.1016/j.scico.2007.11.003

[3] Jan C. Dageförde and Herbert Kuchen. 2019. A compiler and virtual machine for constraint-logic object-oriented programming with Muli. *Journal of Computer Languages* 53 (2019), 63–78. https://doi.org/10.1016/j.cola.2019.05.001

[4] Arjan de Roo, Hasan Sözer, Lodewijk Bergmans, and Mehmet Aksit. 2013. MOO: An architectural framework for runtime optimization of multiple system objectives in embedded control software. *Journal of Systems and Software* 86, 10 (2013), 2502–2519. https://doi.org/10.1016/j.jss.2013.04.002

[5] Jasper Denkers, Marvin Brunner, Louis van Gool, and Eelco Visser. 2021. Configuration Space Exploration for Digital Printing Systems. In *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13085)*, Radu Calinescu and Corina S. Pasareanu (Eds.). Springer, 423–442. https://doi.org/10.1007/978-3-030-92124-8_24

[6] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[7] Karl Anders Ericsson and Herbert Alexander Simon. 1993. Protocol analysis: Verbal reports as data, Rev. (1993).

[8] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages?

[9] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology* 11, 2 (2002), 256–290. https://doi.org/10.1145/505145.505149

[10] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 444–463. https://doi.org/10.1145/1869459.1869497

[11] Sarmen Keshishzadeh, Arjan J. Mooij, and Mohammad Reza Mousavi. 2013. Early Fault Detection in DSLs Using SMT Solving and Automated Debugging. In *Software Engineering and Formal Methods - 11th International Conference, SEFM 2013, Madrid, Spain, September 25-27, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8137)*, Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti (Eds.). Springer, 182–196. https://doi.org/10.1007/978-3-642-40561-7_13

[12] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2010. On the impact of DSL tools on the maintainability of language implementations. In *Proceedings of the of the Tenth Workshop on Language Descriptions, Tools and Applications, LDTA 2010, Paphos, Cyprus, March 28-29, 2010 - satellite event of ETAPS*, Claus Brabrand and Pierre-Etienne Moreau (Eds.). ACM, 10. https://doi.org/10.1145/1868281.1868291

[13] Kevin Leo and Guido Tack. 2017. Debugging Unsatisfiable Constraint Models. In *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10335)*, Domenico Salvagnin and Michele Lombardi 0001 (Eds.). Springer, 77–93. https://doi.org/10.1007/978-3-319-59776-8_7

[14] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: Towards a Standard CP Modelling Language. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4741)*, Christian Bessière (Ed.). Springer, 529–543. https://doi.org/10.1007/978-3-540-74970-7_38

[15] Eugen Schindler, Hristina Moneva, Joost van Pinxten, Louis van Gool, Bart van der Meulen, Niko Stotz, and Bart Theelen. 2021. Jetbrains mps as core dsl technology for developing professional digital printers. In *Domain-Specific Languages in Practice*. Springer, 53–91.

[16] Jouke Stoel, Tijs van der Storm, and Jurgen J. Vinju. 2019. AlleAlle: bounded relational model finding with unbounded data. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, Hidehiko Masuhara and Tomas Petricek 0001 (Eds.). ACM, 46–61. https://doi.org/10.1145/3359591.3359726

[17] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. 2014. The MiniZinc Challenge 2008-2013. *AI Magazine* 35, 2 (2014), 55–60.

[18] Emina Torlak and Daniel Jackson. 2007. Kodkod: A Relational Model Finder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedi (Lecture Notes in Computer Science, Vol. 4424)*, Orna Grumberg and Michael Huth (Eds.). Springer, 632–647. https://doi.org/10.1007/978-3-540-71209-1_49

[19] Markus Voelter. 2018. The Design, Evolution, and Use of KernelF - An Extensible and Embeddable Functional Language. In *Theory and Practice of Model Transformation - 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-26, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10888)*, Arend Rensink and Jesús Sánchez Cuadrado (Eds.). Springer, 3–55. https://doi.org/10.1007/978-3-319-93317-7_1

[20] Markus Voelter, Bernd Kolb, Tamás Szabó, Daniel Ratiu, and Arie van Deursen. 2019. Lessons learned from developing mbeddr: a case study in language engineering with MPS. *Software and Systems Modeling* 18, 1 (2019), 585–630. https://doi.org/10.1007/s10270-016-0575-4

[21] Markus Voelter, Sergej Koscejev, Marcel Riedel, Anna Deitsch, and Andreas Hinkelmann. 2021. A Domain-Specific Language for Payroll Calculations: An Experience Report from DATEV. In *Domain-Specific Languages in Practice: with JetBrains MPS*, Antonio Bucchiarone, Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio (Eds.). Springer, 93–130. https://doi.org/10.1007/978-3-030-73758-0_4