Configuration Space Exploration for Digital Printing Systems

Denkers, Jasper; Brunner, Marvin; van Gool, Louis; Visser, Eelco

DOI
10.1007/978-3-030-92124-8_24

Publication date
2021

Document Version
Final published version

Published in
Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Proceedings

Configuration Space Exploration for Digital Printing Systems

Jasper Denkers[1(✉)], Marvin Brunner[2], Louis van Gool[2], and Eelco Visser[1]

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

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

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

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

1 Introduction

Digital printing systems are flexible manufacturing systems, i.e. manufacturing systems that are capable of adjusting their abilities to manufacture different types and quantities of products, without expensive hardware changes. The variety in printing applications stems from both printing (printing on sheets of paper) and finishing (processing collections of printed sheets, e.g. to form a

R. Calinescu and C. S. Păsăreanu (Eds.): SEFM 2021, LNCS 13085, pp. 423–442, 2021.
https://doi.org/10.1007/978-3-030-92124-8_24

book). The *configuration space* for a digital printing system consists of all possible configurations given the system's features and hardware constraints. For producing a booklet of a particular size, a printed stack of sheets can be stitched, it can be folded, and it can be trimmed. Optionally, the sheets can be rotated in an intermediate production step such that a single trimming component can be used for trimming in multiple dimensions. The decisions made for these manufacturing parameters influence important factors such as productivity (production time increases when sheets are rotated) or efficiency (paper is wasted when input sheets are trimmed).

Ideally, control software assists operators in exploring the configuration space. For example, given some available paper and the intent to produce a booklet, the software should automatically derive a viable manufacturing configuration. Such a configuration e.g. comprises the orientation of the input sheets, the number of stitches, and the amount of side and face trimming needed to get the desired end result. In addition, an optimization objective can be relevant while finding a configuration, e.g. minimizing paper waste. The control software and user interfaces of state of the art digital printing systems do not support such automated configuration space exploration. Instead, operators have to provide configurations for finishers manually. A configuration can be simulated; by "executing" the finishing process in software, finishing viability can be checked without wasting resources. Still, it remains a cognitively intensive task for operators to find a valid or optimal configuration.

Finishers are produced by many vendors and integrating them with printers is non-trivial. Such integration involves connecting the control software of the printer and finishers and driving embedded software components. Using a classical modeling and integration approach to support the variety of finishing is suboptimal with respect to development time and maintenance burden. Issues with such a classical approach are the long code-build-test cycle and the large amount of finisher vendors and models that must be supported for many years. The translation of the mechanical specifications into control software code gives rise to additional complexity.

Our objective is to obtain an effective, efficient, and scalable method for modeling finishers and obtaining control software for finishers that support automated configuration space exploration. In this work, we investigate how linguistic abstraction can help to model the configuration space of digital printing systems, and how we can automatically derive environments for configuration space exploration from such specifications.

The global characteristics of finishers make the use of constraint (SMT) solving a natural fit for realizing environments for configuration space exploration. For example, trimming the paper along a certain dimension might impose a specific orientation or transformation in an earlier production step. A constraint-based approach considers its specifications as global and will take into account interdependent system-level constraints when finding solutions, i.e., configurations. A constraint-based model of a finisher contains a representation of the input materials at intermediate locations in the system. However, for modelling

domain objects such as sheets and stacks, abstraction mechanisms such as classes are not naturally available in SMT modelling. An SMT model of a finisher requires low-level encoding of the properties of the materials at all locations. Therefore, expressing finishers in SMT by hand is tedious, error prone, and is not in terms of domain concepts. Additionally, an SMT model of a finisher is complex to understand and difficult to maintain.

In this paper, we present CSX, a domain-specific language for the high-level declarative specification of finishers. The language supports specification of input materials, configuration parameters, output products, and finishing constraints in terms of domain concepts. The CSX IDE supports the development and checking of specifications and the automated derivation of an environment for configuration space exploration by operators of the finishers.

CSX provides a domain-specific interface to SMT solving by abstracting and structuring over low-level properties. We translate specifications to the SMT domain and use existing solvers to find solutions at the level of properties and finishing parameters. A solution in the SMT domain corresponds to a valid configuration. Unsatisfiability at the SMT level indicates an empty configuration space, i.e., no finishing possibilities. By mapping SMT solutions back to the specification level, we can interpret CSX specifications in multiple modes: checking whether a configuration is valid, finding an (optimal) configuration, and validating specifications. By caching invocations of the solver in the IDE, response times are improved which leads to an interactive editing experience.

The approach of specifying a finisher with CSX and deriving control software has similarities with the approach of simulation in control software. Both approaches take representations of the products being produced at intermediate locations in the devices. However, while simulation involves an operational and sequential application of transformations on objects, a constraint-based approach considers the devices globally. CSX improves over simulation in the sense that it derives environments that can search for (optimal) configurations in an automated way, taking system-global interdependencies into account.

We evaluated the design and implementation of CSX by modelling two finishers: a perfect binder and a booklet maker. In the process of modelling these devices, we have experimented with various encodings. For both cases, we benchmark the configuration space exploration performance for several scenarios.

Contributions. To summarize, the contributions of this paper are the following:

- We have developed CSX, a declarative language for the specification of finishers at the conceptual level of the domain. We interpret CSX specifications for several modes of configuration space exploration: checking whether configurations are valid, finding optimal configurations under objectives, and interactively validating specifications.
- We define a denotational semantics of CSX in terms of SMT constraints that serves as an interface to solvers that can be used to find models in order to check inhabitance of a specification and to explore the configuration space of the specified finisher.

- We realize a programming environment for CSX that integrates an SMT solver as back-end and that presents solutions in terms of the specification.
- We evaluate CSX by specifying two types of finishers: a perfect binder and a booklet maker. For these cases, we benchmark the performance for a configuration space exploration scenario with and without optimization.

2 Finishers in the Digital Printing Domain

In this section, we discuss the domain of digital printing systems with finishers. Complete printing systems for e.g. producing books include, in addition to printing itself, finishing capabilities. Finishing comprises the processing of printed sheets of paper into end products. For example, a stack of printed sheets could be stapled, folded, and trimmed to result into a booklet; stapling, folding, and trimming are finishing operations. Finishing devices need to be integrated with the printing system for realizing an integrated end-to-end experience for the print system end-users (i.e. operators in print shops).

The turnaround time of integrating finishers with printers is high because of multiple challenging aspects. First, finishers are often produced by external vendors and communication is mostly documentation based and thus requires interpretation, reviews, implementation, and testing. Second, obtaining good system behavior requires mechanical, electrical and software interfaces to be matched well between the printer and finisher. Third, total aspects such as reliability are the result of all the mentioned interfaces to be well designed. Considerable testing time is needed to confirm reliability.

Creating control software that is user-friendly for operators is difficult and requires a lot of manual programming. This is because of the high variability and many configuration parameters in print and finishing systems. A typical print and finishing system has more than 200 accessible parameters for the operator, that are also interdependent. Because the whole production process is a sequence of production steps, choices that you have to make in the beginning influence the steps later on. From the product line perspective, the control software supports tens of different finisher types, that each of them can have more than 100 commercial variations. For all variations, the parameters that are accessible for operators can vary.

Ideally, operators can use the combination of a printer with finishers as an end-to-end solution instead of having to configure each device separately. Additionally, optimization capabilities are also useful when considering the system as a whole. For example, an operator would like to produce booklets with the available resources and while minimizing paper waste or while optimizing productivity. If the different configuration possibilities impose a tradeoff between e.g. resource consumption and productivity, an operator should be able to make a motivated choice with ease, i.e., without thinking about and manually trying out many combinations of configuration parameters.

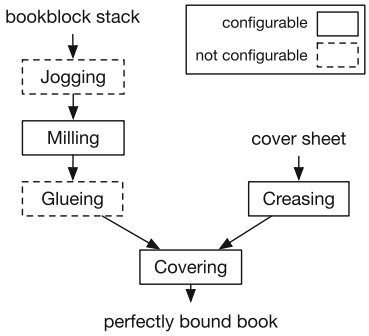

Fig. 1. Schematic view of the perfect binding book producing process. Only milling, creasing, and covering are configurable and therefore impact the configuration space. Jogging and glueing are automatically configured by the device itself.

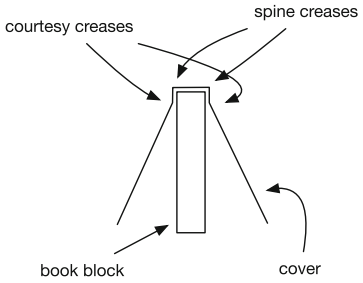

Fig. 2. A perfectly bound book viewed from the top. Spine creases result into a sharper fold, reduce wrinkles, and improve the fit of the cover around the bookblock. Courtesy creases ease opening the front and back part of the cover. Glue in the spine holds the bookblock sheets and cover together.

2.1 Perfect Binding

As an example, we discuss a *perfect binder*: a finisher that produces books by binding a stack of sheets with glue and by covering the bookblock in a cover sheet. A perfect binder typically has two inputs: one for the stack of sheets that form the book block and one for the cover sheets. Figure 1 shows the perfect binding process. Figure 2 depicts the components of a perfectly bound book, viewed from above.

After collecting a stack of sheets, jogging makes sure the stack of sheets becomes aligned in a corner of the spine. Then, a clamp grasps the bookblock under pressure. Next, a few millimeters of paper are milled along the spine edge to prepare the spine for application of glue. Milling makes the paper along the spine rough, improving adherence of the glue. Then, the spine travels through a bath of heated glue.

Separately, cover sheets are prepared before being bound around the bookblock. The preparation consists of creasing, i.e., applying pressure on the paper to ease folding of the paper later. Two creases are applied at the location of the cover that end up along the edges of the spine of the book. These creases improve the fit of the cover along the spine of the book block, supporting a tight fold around the spine. Additionally, two courtesy creases are applied on the cover. Courtesy creases are applied on the front and back of the resulting book to support the folding of the cover sheet. Note that courtesy creases are applied at the opposite side as the spine creases, as they are used for folds in opposite directions.

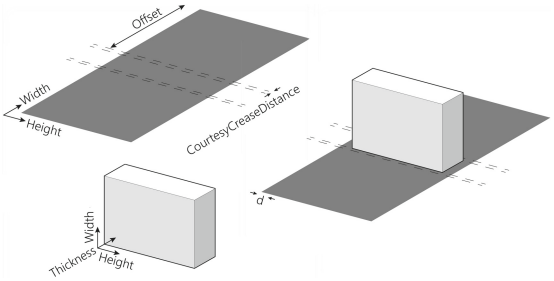

Fig. 3. Components of a perfectly bound book (cover and bookblock) and the dimensions as how we use them in the CSX specification.

After preparing the bookblock and cover, the covering occurs. The bookblock with glue is positioned in the center of the cover sheet. The cover sheet is folded around the bookblock and fixed with a clamp. After a delay for the glue to solidify, the book is released. In practice, the resulting book could be processed further in a cutting machine to trim along the edges of the book and cover to result into a nice book.

Perfect binders are flexible in the books they can produce, e.g. in terms of sheet size or book thickness. Not all flexible manufacturing steps have impact on the configuration space. For example, jogging and glueing occur automatically and are configured by the device itself based on measurements. Other settings such as the milling depth and positioning of the bookblock on the cover are of interest to the operator and therefore do impact the configuration space; e.g. more milling might increase the overall production time.

3 CSX

The key idea of CSX is that we model objects such as sheets and stacks and that we specify symbolic values, i.e. instances, for these objects at several intermediate steps in the finishing process. By adding constraints and indicating configuration parameters, a specification defines the configuration space of a device. In CSX we also describe *jobs*, i.e., (partial) descriptions of the production process in terms of the production objects and parameters. We achieve configuration space exploration by synthesizing configurations from a configuration space for a given job.

CSX is declarative: a specification in the language describes behavior and configuration spaces of finishers. A CSX specification does not describe algorithms to compute configurations. Specifications include relations between objects at locations in the systems. We use the language to model devices as sequences of components that perform actions. Components instantiate generic, reusable actions. Actions establish a relationship between snapshots of objects in the finishers and thus, transitively, devices define a relation between all snapshots of the products being produced. Parameters in actions represent a dimension of

```
type Sheet {
 width: int, [width > 0],
 height: int, [height > 0]
}
type Stack {
 width: int, [width > 0],
 height: int, [height > 0],
 thickness: int, [thickness > 0],
 volume = width * height * thickness
}
type PerfectBoundBook {
 book: Stack,
 frontCover: Sheet, backCover: Sheet
}
```

```
type CreasedSheet {
 sheet: Sheet,
 spineFront: Crease,
 spineBack: Crease,
 courtesyFront: Crease,
 courtesyBack: Crease
}

type Crease {
 // Offset:
 off: int, [off ≥ 0],
 // Direction: 0 = down, 1 = up
 dir: int, [dir == 0 or dir == 1]
}
```

Fig. 4. The specification of types for the example perfect binder in CSX. Dimensions are in 0.1mm.

```
action ToMill(in: Stack, out: Stack) {
 parameter millingDepth: int
 [millingDepth ≥ 0] [out.width == in.width - millingDepth]
 [out.height == in.height] [out.thickness == in.thickness]
}
action ToCrease(in: Sheet, out: CreasedSheet) {
 [out.sheet.width == in.width] [out.sheet.height == in.height]
 [out.spineFront.dir == 0] [out.spineBack.dir == 0]
 [out.courtesyFront.dir == 1] [out.courtesyBack.dir == 1]
 // Ensure a minimum distance between creases
 parameter minDist: int [minDist > 0] [out.courtesyFront.off ≥ minDist]
 [out.spineFront.off ≥ out.courtesyFront.off + minDist]
 [out.spineBack.off ≥ out.spineFront.off + minDist]
 [out.courtesyBack.off ≥ out.spineBack.off + minDist]
 [out.courtesyBack.off ≤ in.width - minDist]
}
action ToCover(cover: CreasedSheet, book: Stack, out: PerfectBoundBook) {
 parameter d: int [0 ≤ d and d ≤ cover.sheet.height - book.height]
 [out.book.width == book.width] [out.book.height == book.height]
 [out.book.thickness == book.thickness]
 [out.frontCover.width * 2 == cover.sheet.width - book.thickness]
 [out.frontCover.height == cover.sheet.height]
 [out.backCover.width * 2 == cover.sheet.width - book.thickness]
 [out.backCover.height == cover.sheet.height]
 [cover.spineFront.off * 2 == cover.sheet.width - book.thickness]
 [cover.spineBack.off * 2 == cover.sheet.width + book.thickness]
 parameter courtesyCreaseDist: int
 [cover.courtesyFront.off * 2 ==
   cover.sheet.width - book.thickness - courtesyCreaseDist * 2]
 [cover.courtesyBack.off  * 2 ==
   cover.sheet.width + book.thickness + courtesyCreaseDist * 2]
}
```

Fig. 5. The specification of actions for the example perfect binder in CSX. See Fig. 3 for the dimensions used in this specification.

```
device ExamplePerfectBinder {
 location bookIn : Stack location coverIn : Sheet
 [1000 ≤ bookIn.height and bookIn.height ≤ 3000]
 [2000 ≤ bookIn.width and bookIn.width ≤ 5000]
 component toMill = ToMill(bookIn, milledBook) {
  [millingDepth ≤ 30] // Max 3mm of milling
  [bookIn.thickness < 170] // Max 17mm book thickness
 }
 location milledBook : Stack
 component toCrease = ToCrease(coverIn, creasedCover) {
  [minDist ≥ 50] // At least 5 mm between creases
 }
 location creasedCover : CreasedSheet
 component toCover = ToCover(creasedCover, milledBook, out) {}
 location out : PerfectBoundBook
}
```

Fig. 6. The specification of the example perfect binder device in CSX.

configuration that is of interest to operators of the devices. Constraints restrict instances of types and restrict the behavior of actions and devices, reducing the configuration space. We will now introduce the language concepts in more detail based on a specification for an example perfect binder such as described in Sect. 2.

Defined *types* are records of properties that model objects at locations in a device. In Fig. 4, we define several types for the example perfect binder. Dimensions (widths, heights, lengths, distances) are modelled with integers with a precision of 0.1mm, such that an integer value of 10 stands for a length of 1mm. Types contain *defining properties* that are of a primitive type (boolean or integer) or of a defined type such that types can be nested. The nesting of types may not contain a cycle. Types optionally contain *constraints* and *derived properties*. Constraints restrict the inhabitants of a type. In Fig. 4, the constraints (between square brackets) e.g. restrict sheets to have positive non-zero width and height. Derived properties are shorthands for expressions over other properties. Defining properties are required to instantiate a type. Derived properties are not required to instantiate a type and their values can be derived from other properties. A derived property expression may refer to the type's properties and to other derived properties, but derived properties may not contain cyclic references. In Fig. 4, `Stack` has a derived property `volume` which is defined in terms of defining properties.

Actions define a relation between locations. In Fig. 5, we define several actions for the example perfect binder. The body of an action definition contains parameters and constraints that indicate the relations between its parameters.

Devices are sequences of *components* connected through *locations*. Components instantiate actions and can restrict or specify behavior further by adding constraints. Thus, action behavior is defined separately from specific instantiations in components. Therefore, actions are generic and potentially reusable between different device specifications. Limitations of a particular instance of

an action in a device can be specified by adding constraints to the component. In Fig. 6 we define a perfect binder device by instantiating several actions in components and by connecting them through the locations.

3.1 Configurations and Jobs

A configuration for a device is a value assignment to all locations and parameters. A valid configuration is a configuration that conforms to the constraints of the types of the locations, the actions, the components, and the device itself. In practice, an operator is only interested in the values for the input and output locations, and not in the intermediate locations.

A job is an expression of intent for which a configuration needs to be found. Whereas configurations are a complete specification of locations and parameters, we could see jobs as a partial configuration. For example, a job could define the input and the output of the finisher. The remaining parts of the configuration, i.e. the finishing parameters, need to be derived in order to instruct the finisher to realize the intent of the job. Different usage scenarios of a device lead to different jobs and approaches to configuration.

3.2 Exploration and Validation

The CSX language supports configuration space exploration, which includes leveraging exploration at the specification level for validation. Given the specification of a device, the language supports describing scenarios for testing devices by asserting expectations on configuration spaces.

The following test scenario validates that the correct cover dimensions are chosen for a particular input bookblock and desired output perfectly bound book:

```
scenario device ExamplePerfectBinder
 config bookIn = Stack(2125,2970,50)
 config out = PerfectBoundBook(Stack(2100,2970,50), Sheet(2100,2970), Sheet(2100,2970)) {
  [coverIn.width == 2100 + 2100 + 50]
  [coverIn.height == 2970]
  [toMill.millingDepth == 25]
}
```

The body of the scenario contains expectations (between square brackets) on its configuration space. In particular, it validates the cover dimensions that must be chosen. Since the configuration space could contain multiple configurations, expectations should only validate common properties of the configuration space and not on individual configurations.

Scenarios can optionally specify an objective. *Objectives* indicate a dimension for optimization of a property of the system, typically expressed using derived properties. Potentially relevant objectives are e.g. maximizing throughput, minimizing energy consumption, or minimizing resource waste. Alternatively, scenarios with optimization can characterize the device. For example, based on the

following scenario a scenario can be found for the largest book that the perfect binder can produce:

```
scenario device ExamplePerfectBinder
         maximize out.book.volume
```

4 Denotational Semantics

Because of the declarative characteristic of CSX, a translation to SMT constraints is natural. In this section, we define the denotational semantics of CSX that expresses a translation of CSX specifications to SMT constraints. Figure 7 contains the denotational semantics of CSX with the denotation expressed in MiniZinc [9,13] definitions. Because we use MiniZinc in the implementation of CSX (Sect. 5), we also use it as syntax for the denotation. The MiniZinc grammar can be found online[1].

The intuition behind the translation is that the properties of locations and the parameters of components are mapped to constraint variables. Additionally, all CSX-defined constraints translate to corresponding constraints in MiniZinc. The translation is from the perspective of a device, making use of type and actions definitions of the CSX specification of which the device is part.

The translation starts with the Device rule, generating MiniZinc definitions for members of the device: locations, components, and device-level constraints. The translation is defined under the context of a namespace N, starting with the empty namespace. The naming scheme for constraint variables follow their corresponding hierarchical position in the CSX specification. Since the translation is for a single device, we do not have to prefix the namespace with the device name.

A location translates into variables for its properties and into constraints to restrict its inhabitants (Location). Locations are always of a user-defined type. Each property of the type translates to variables. If the property is of primitive type, the translation is a variable of this primitive type (DefProp-PrimType)). If the property is of a user-defined type, the translation is the translation of its nested properties in the namespace of the property (DefProp-DefType).

The Comp rule defines the translation for a component, i.e. an action instantiation. The action's parameters translate into variables in the namespace of the component (Param). Both the action and the component can define constraints (E_i^A and E_i^C, respectively). These constraints are mapped to corresponding MiniZinc constraints. Since the action's constraints are defined on the action's location parameters, and the action gets instantiated with specific location arguments, renaming is required. The translation defines R: a mapping from the location's parameter names to the component's location argument names. We only use the renaming for translating references to locations from constraints defined in the action definition.

[1] https://www.minizinc.org/doc-2.5.5/en/spec.html?highlight=grammar#spec-grammar.

$[\![S']\!]_{S,N,R} = M$ Specification part S' of S translates to M in namespace N with location renaming R

$N = [x_1, x_2, \ldots, x_n]$ Namespace N consisting of parts x_1 to x_n

$R = \{\ldots, L_i \to L_i', \ldots\}$ Renaming of location names L_i to L_i'

$name([x_1, x_2, \ldots, x_n]) = x_1\_x_2\_\ldots\_x_n$ Identifier for namespace $[x_1, x_2, \ldots, x_n]$

Locations L, components C, constraints E, defining properties P, types T, action parameters PM.

Devices

$$[\![\texttt{device } d \ \{\ L_1 \ldots L_n, C_1 \ldots C_m, E_1 \ldots E_q, \ldots\ \}]\!]_{S,[],\emptyset} = \bigcup_{i=1}^{n} [\![L_i]\!]_{S,[],\emptyset} \cup \bigcup_{i=1}^{m} [\![C_i]\!]_{S,[],\emptyset} \cup \bigcup_{i=1}^{q} [\![E_i]\!]_{S,[],\emptyset} \quad \text{(Device)}$$

Locations

$$\frac{\texttt{type } T \ \{\ P_1{:}T_1 \ldots P_n{:}T_n, E_1 \ldots E_m, \ldots\ \} \in S}{[\![\texttt{location } L : T]\!]_{S,[],\emptyset} = \bigcup_{i=1}^{n} [\![P_i{:}T_i]\!]_{S,[L],\emptyset} \cup \bigcup_{i=1}^{m} [\![E_m]\!]_{S,[L],\emptyset}} \quad \text{(Location)}$$

$$\frac{T \in \{\texttt{int}, \texttt{bool}\}}{[\![P{:}T]\!]_{S,N,\emptyset} = \texttt{var } T \ : \ name(N +\!\!+ [P]) \ ;} \quad \text{(DefProp-PrimType)}$$

$$\frac{\texttt{type } T \ \{\ P_1{:}T_1 \ldots P_n{:}T_n, E_1 \ldots E_m, \ldots\ \} \in S}{[\![P{:}T]\!]_{S,N,\emptyset} = \bigcup_{i=1}^{n} [\![P_n{:}T_n]\!]_{S,N+\!\!+[P],\emptyset} \cup \bigcup_{i=1}^{m} [\![E_m]\!]_{S,N+\!\!+[P],\emptyset}} \quad \text{(DefProp-DefType)}$$

Components

$$\frac{\begin{array}{c}\texttt{action } A(L_1{:}T_1^L \ldots L_n{:}T_n^L) \\ \{\texttt{parameter } PM_1 : T_1^P \ldots \texttt{parameter } PM_m : T_m^P, E_1^A \ldots E_q^A, \ldots\} \in S \\ R = \{L_1 \to L_1', \ldots, L_n \to L_r'\}\end{array}}{\begin{array}{c}[\![\texttt{component } C \texttt{ = } A \texttt{ (} L_1' \ldots L_r' \texttt{) } \{\ E_1^C \ldots E_s^C\ \}]\!]_{S,[],\emptyset} = \\ \bigcup_{i=1}^{m} [\![\texttt{parameter } PM_m : T_m^P]\!]_{S,[C],\emptyset} \cup \bigcup_{i=1}^{q} [\![E_i^A]\!]_{S,[C],R} \cup \bigcup_{i=1}^{s} [\![E_s^C]\!]_{S,[C],\emptyset}\end{array}} \quad \text{(Comp)}$$

$$\frac{T \in \{\texttt{int}, \texttt{bool}\}}{[\![\texttt{parameter } PM{:}T]\!]_{S,N,\emptyset} = \texttt{var } T{:}name(N +\!\!+ [PM]) \ ;} \quad \text{(Param)}$$

Constraints & References

$$[\![\ \texttt{[} e \texttt{]}\ ]\!]_{S,N,R} = \texttt{constraint } [\![e]\!]_{S,N,R}; \quad \text{(Constraint)}$$

$$\frac{x \text{ is a defining property or parameter}}{[\![x]\!]_{S,N,R} = name(N +\!\!+ [x])} \quad \text{(DefProp-Ref/Param-Ref)}$$

$$\frac{x \text{ is a location} \qquad x \to x' \notin R}{[\![x]\!]_{S,N,R} = name(N +\!\!+ [x])} \quad \text{(Location-Ref)}$$

$$\frac{x \text{ is a location} \qquad x \to x' \in R}{[\![x]\!]_{S,N,R} = name(N +\!\!+ [x'])} \quad \text{(ActionLocation-Ref)}$$

$$\frac{x \text{ is a derived property with body } e}{[\![x]\!]_{S,N,R} = [\![e]\!]_{S,N,R}} \quad \text{(DerProp-Ref)}$$

$$[\![e.x]\!]_{S,N,R} = [\![e]\!]_{S,N,R} + \_x \quad \text{(Proj)}$$

Fig. 7. Denotational semantics of CSX, expressed in MiniZinc. We have omitted the rules for literals and arithmetic for brevity; they map one-to-one. $+\!\!+$ is namespace concatenation. $+$ is identifier concatenation.

The expressions that are used to define constraints, except references and projection, map mostly one-to-one to their MiniZinc counterparts. For references and projection, we consider several cases. A reference to property or parameter (DefProp-Ref/Param-Ref) translates to a name for x in the context. For example, a reference of x in namespace $[a, b]$ will result in the denotation into a reference to name `a_b_x`. For projection (Proj), we recursively translate the base expressions into a name and concatenate the projected name.

For a location reference, we consider two cases. Location references from outside actions translate similarly as regular references (Location-Ref). Location references within actions refer to location parameters, while the actions are instantiated with location arguments from a device. Therefore, for such location references, we replace the location parameter name by the argument name for which it is instantiated (ActionLocation-Ref).

Types, actions, and devices can have derived properties. These only translate into constraints if they are referenced, i.e. by replacing the reference with the body of the derived property and by propagating the namespace and location renaming (DerProp-Ref). For the definition of derived properties, no translation takes place. The definition of derived properties are ignored by ... in the specification.

Solutions found for the MiniZinc denotations are related to valid configurations for CSX specifications, and we can translate such solutions back to CSX Specifications.The correspondence between location properties and component parameters in CSX and MiniZinc is defined by the naming scheme used in the denotation, and mapping them back is thus straightforward.

5 Implementation

In this section we describe how we obtain a usable integrated development environment (IDE) for CSX by integrating an implementation of the language with configuration space exploration and interactive validation. The IDE contains components for parsing, syntax highlighting, code completion, name binding and type checking, and interactive reporting of static semantics violations. The CSX validation constructs are interpreted interactively and invalid assertions are marked on the specification.

We have implemented the CSX language using Spoofax [7], a language workbench [5] that provides infrastructure for designing, implementing, and deploying DSLs by means of declarative specification of language aspects using meta-DSLs. We define the syntax of CSX in SDF3 [11], a meta-language for multi-purpose syntax definition. From the CSX syntax definition, SDF3 automatically derives a parser, pretty printer, syntax highlighting, and syntactic code completion. The parser yields abstract syntax trees (ASTs) on which we first apply desugaring. Desugaring e.g. involves propagating the properties of a scenario to the tests within that scenario. The desugared ASTs are input to the static analysis and further transformations. We specify desugaring and other transformations using the Stratego [2] meta-language. Based on the language specification, Spoofax automatically generates an IDE for the language.

We define the CSX static semantics in NaBL2 [1,10]. NaBL2 is a meta-language for specifying static semantics for languages from which name binding and type checking is automatically derived. Static semantic violations are reported interactively in the IDE. For CSX, this could be invalid composition of components in a device or incorrect type checking of constraint expressions. Interactive reporting of errors assists users of the language during specification writing.

In addition to the automated derivation of name binding and type checking, we implement analysis for other well-formedness conditions. If well-formedness checking succeeds, the result is a desugared AST that is annotated with name binding and typing information. The name binding information is used to check non-cyclic references of defining properties and derived properties, i.e., by following references of properties and checking whether those do not contain cycles.

To realize configuration space exploration, we implement a translation of CSX specifications to SMT constraints for which we can use existing solving techniques. In particular, we translate CSX to the MiniZinc constraint modelling language [9,13]. MiniZinc is solver-independent, which enables us to use multiple solvers as a backend for CSX. In particular, we use solvers with the theories of linear arithmetic and optimization modulo theories.

We implement the translation from CSX to MiniZinc as an AST-to-AST transformation using Stratego. In addition to the syntax definition of CSX, we have also defined the syntax of MiniZinc in Spoofax with SDF3[2]. The syntax definitions of both languages generate an AST schema on which we define the Stratego transformation. After transforming a parsed CSX AST to a MiniZinc AST, the MiniZinc pretty printer generates concrete MiniZinc syntax from the AST.

The translation uses information from name binding and type analysis. This is necessary for references and projection expressions. By using name binding and typing information, the distinction between references to properties, parameters, locations, and action locations can be made to generate the correct reference on the MiniZinc level.

We integrate solving of constraint models by calling MiniZinc from Stratego through integration with Java. Stratego provides an API for integrating transformations with custom Java code. We implement such a custom transformation and use a Java program to call the MiniZinc command-line interface. The Java program is called with as input the generated MiniZinc model. The Java program parses the textual solving result that is returned by MiniZinc and returns it as a list of variable binding. In the Stratego code, for the interpretation of configurations, we evaluate expressions and lookup values for references by following the same naming schema as in the translation semantics. After replacing the referenced properties and parameters by their values on the constraint level, the evaluation of expressions remains regular expression evaluation. As a result, we have a configuration space exploration pipeline from interpreting specifications

[2] https://github.com/metaborgcube/metaborg-minizinc.

using constraint solving with the solution mapped back to the specification level as a configuration.

The configuration space exploration pipeline serves two purposes in the IDE: test evaluation and inhabitance checking. For test evaluation, the configuration space of the device that is selected in the scenario is translated to MiniZinc and passed as an input to the pipeline. Additional constraints are added to reduce the configuration space, e.g. to configure the input or output location values, or parameters as specified in the scenario. If the scenario contains an objective, the objective is also mapped to MiniZinc and provided as input to the pipeline. The configuration that is returned by the pipeline is used to evaluate test expectations. This evaluation is done by a basic interpreter that evaluates expressions which should result into true. The expressions can contain references to parameters and location properties, and based on the name binding information the references are mapped to the corresponding value from the configuration. For failed test expectations we report an error which is marked with red underlining on the original specification using origin tracking [4].

The evaluation of tests and reporting of results is triggered in the IDE on file changes, resulting into an interactive experience. Additionally, the experience is improved by providing information while hovering over references to locations, properties, and parameters in test expectations. The same interpretation approach as for test expectations is used to evaluate the expression being hovered over and the value is presented in a popup, giving the user insight in the configuration that is found.

Similar to the treatment of scenarios, inhabitance checks are triggered on file changes. The pipeline is triggered for each type, action, and device using the translations semantics. For inhabitance checking of a type, we translate a random instance of that type to SMT. For an action, we instantiate it with instances for all its parameters. Instead of finding a configuration for it, for inhabitance checking we only check satisfiability on the constraint level. If the pipeline concludes in satisfiability, we report an error on the corresponding construct to indicate that the construct is not inhabited.

To prevent unnecessary checking of inhabitance and evaluation of tests, we use simple caching of analysis results with ASTs of the subjects as the caching key. If a type definition AST has not changed, it does not have to be checked again for inhabitance. If a scenario has not changed, it does not have to be evaluated again.

While we have described the realization of a programming environment for CSX specifications, the eventual goal of CSX is to deploy control software to finishers. Figure 8 gives an overview of how configuration space exploration with CSX would with fit in a realistic setting. The configuration space exploration component would be integrated with a software component, implemented using a general-purpose language, that provides a UI and that instructs low-level embedded software components.

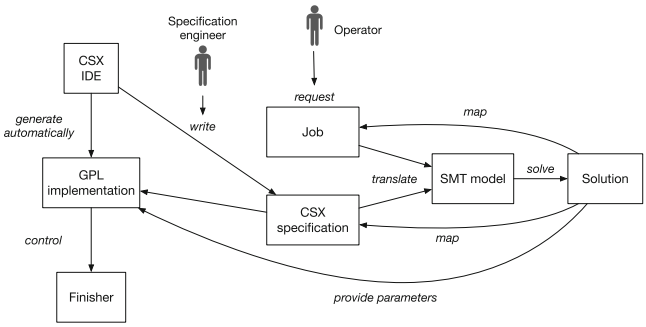

Fig. 8. An architecture for applying CSX in control software. GPL stands for general purpose programming language, such as C# or Java.

6 Evaluation

We evaluate CSX by modelling two realistic cases, a perfect binder and a booklet maker, and by benchmarking the configuration space exploration for a scenario with and without optimization. The perfect binder case corresponds to the example of Sect. 3. In the scenario without optimization, CSX derives the required input cover given an input bookblock and a desired output. In the scenario with optimization, CSX finds a configuration for the smallest size book the finisher can produce. The bookletmaker case concerns a finisher that performs rotating, stitching, folding, and trimming in order to produce a booklet from a stack of sheets. In the scenario without optimization, CSX finds the action parameters given an input and output. In the scenario with optimization, CSX finds a configuration that minimizes paper waste given only the desired output. Both specifications are based on realistic cases present at Canon Production Printing B.V.

By writing scenarios in the language, we can interactively validate the specification within the IDE. Initially loading a specification can take a few seconds: a specification typically consists of multiple type definitions, action definitions, a device definition, and several scenarios. For the type, action, and device definitions, inhabitance checking is triggered, which for each check leads to an invocation of the SMT solver. Additionally, for each scenario the solver is invoked. The caching of invocations of the solver decreases response times after a change, making the IDE usable in an interactive way. For example, inhabitance for a type will not be re-checked if only a test scenario changes.

We set up a benchmark which makes use of Spoofax core, i.e. the core of Spoofax which enables integration of language components with Java, such that we can only execute the relevant part of the pipeline in the benchmark. For benchmarking, we use the JMH framework[3]. We executed the benchmarks on a server with two 32-core processors with a base frequency of 2.3 GHz and 256

[3] https://openjdk.java.net/projects/code-tools/jmh/.

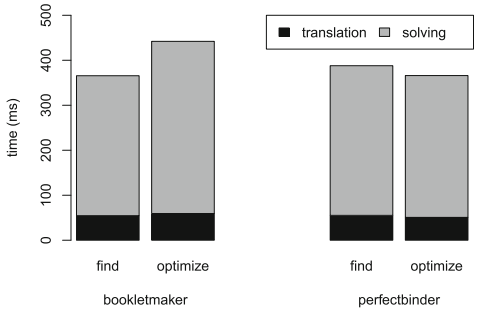

Fig. 9. The benchmarking results on a perfect binder and a booklet maker for a scenario of finding a configuration and for finding an optimal configuration.

GB RAM, running Ubuntu 20.04, using OpenJDK version 1.8.0_275-b01. From experimentation it appeared that the ORTools solver[4] had best performance, and therefore we use this solver in the benchmarks. We use MiniZinc version 2.5.5 and ORTools version 9.0. We measure 10 iterations and average the result. In the benchmarks, we separately measure the translation time and solving time. We leave out parsing, name binding and type checking time, as they are minimal compared to translation and solving time.

Figure 9 shows the benchmarking results. For each scenario, solving time is in the order of 100's ms. We consider sub-second performance as practical and therefore conclude that CSX's performance for the two cases we consider has practical performance for finding (optimal) configurations.

For specifying these devices in CSX, we have chosen a model of objects (sheets, stacks) with a certain level of detail. The bookletmaker and perfect binding cases translated in the SMT level into 32 and 29 variables and 56 and 58 constraints, respectively. Although we achieve useful configuration space exploration for these scenarios, it could be that in practice more detail has to be added to the model, which could also influence solving performance. By deploying CSX at Canon Production Printing B.V., we aim to further evaluate whether CSX is adequate in modeling and integrating the full product line of finishers available and evaluate its usability for domain experts.

7 Related Work

We discuss related work that uses constraint solving in the backend of high-level specification or domain-specific languages for realizing static analyses, validation, verification, consistency checking or synthesis.

Keshishzadeh et al. use SMT solving for validation of domain-specific properties to achieve fault detection early in the software development cycle. In particular, they develop a DSL with industrial application in a case on collision prevention for medical imaging equipment [8]. The approach includes delta

[4] https://developers.google.com/optimization.

debugging, i.e., an approach to trace causes of property violations and report them back to the specification in a systematic way. The work is related to CSX because it also uses SMT solving in the backend of a domain-specific language.

Voelter et al. use SMT solving with the Z3 solver for advanced error checking and verification in the KernelF language [16], a reusable functional language for the development of DSLs. Voelter et al. apply SMT solving successfully in a DSL on a case study for the domain of payroll calculations [17], i.e. for statically checking completeness and overlap of domain-specific switch-like expressions. Similarly to CSX, in this work SMT solving is used in the backed of a domain-specific language for realizing static analyses. While the application of SMT was successful in the domain-specific case, the authors report difficulties in applying SMT solving generically in KernelF. The authors plan to develop a successor to KernelF that is realized with SMT solving completely.

Constraint solving in feature models solves a different problem than CSX. Feature models describe systems as compatible compositions of features or software components; finding/checking feature compositions occurs "statically" from which a software artifact can be derived. CSX specifications express physical properties of finishers; finding configurations occurs "dynamically" (at run time) to find instances of the manufacturing process. This goes all the way down to the "semantic" level, e.g. by using sheet dimensions and the location of fold edges instead of only an abstract feature that enumerates the kinds of folds a device can do. Feature modelling is useful in the finishing context e.g. to derive which devices are necessary for a production route for booklets. In CSX, we assume the production route is known.

Relational model finders are related to CSX in the sense that they map high-level specifications to constraints and map solutions back to the specification level. Alloy [6] is a specification language that applies finite model finding to check formal specifications of software. Alloy is backed by KodKod [15], a relational model finder for problems expressed using first order logic, relational algebra, and transitive closures. In contrast to CSX, KodKod does not offer support for reasoning over data nor for optimization objectives. In CSX, the nature of specifications is not relational: manufacturing paths are fixed and we consider snapshots of the product being manufactured at different steps in the process.

AlleAlle [12] adds support for first-class data attributes and optimization to relational model finding. Similar to KodKod, Stoel et al. consider AlleAlle as an intermediate language. AlleAlle and CSX are related in the sense that both approaches take the data of problems into account and use SMT solving for model finding. While AlleAlle is an intermediate language generally targeting relational problems, CSX is a more domain-specific language in which relations are not a first class concept. Similar to CSX, for AlleAlle it is unclear yet how to map reasons for unsatisfiability that are found in the constraint level back to the specification level.

Rosette [14] is a solver-aided programming language that supports verification, debugging, and synthesis. Rosette extends the Racket language with support for symbolic values that stand for e.g. an arbitrary integer value. Such values

translate to a constraint variable in the runtime. Rosette realizes verification and synthesis in the runtime by integrating its symbolic virtual machine with SMT solvers. Whereas in Rosette selected variables are replaced by symbolic values, in CSX all variables in the specification translate to constraint variables. Rosette is a general language tailored to program verification and synthesis whereas CSX is focused on a particular domain, i.e. manufacturing systems, although we have only experimented with CSX in the digital printing domain.

Muli [3] is a constraint-logic object oriented language that integrates constraint solving with object oriented programming in the Java programming language. Muli extends Java's syntax with the `free` keyword for indicating symbolic values that translate to constraint variables in the runtime. Fragments of programs that are considered as search regions are executed non-deterministically, searching for concrete values for the constraint variables. The Muli runtime is based on a symbolic Java virtual machine that integrates constraint solvers. Muli only supports primitive types as constraint variables. Support for arrays and objects as constraint variables is listed as future work. CSX does support search on non-primitive types such as user-defined record types. Similar to how support for arrays is desired for Muli, support for lists is desired for CSX, but that is future work. Muli differs from CSX in the sense that Muli preserves the Java syntax and, by doing so, serves as a general purpose programming language, whereas CSX introduces a new domain-specific language. In contrast to Muli, CSX supports optimization.

8 Conclusions

We have presented CSX, a language and method for high-level declarative specification of finishers and their configuration spaces. We have developed a translation of CSX to SMT constraints which enables us to use constraint solving to find (optimal) configurations for finishers. We have presented an implementation of the CSX programming environment, including support for well-formedness checking, inhabitance checking, and interactive configuration space exploration. Our benchmarks show that, on two realistic cases, CSX has practical sub-second performance in finding configurations for scenarios with and without optimization.

Future work. Our focus has been on finding a domain abstraction for configuration space exploration applied in the digital printing domain for finishers. While we have designed the language in collaboration with control software engineers, we plan to further evaluate CSX by deploying it at Canon Production Printing B.V. By doing so, we can further evaluate the adequacy of CSX in covering the full product line of finishers. Additionally, we plan to evaluate the language in terms of usability for control software engineers and in terms of validatability by mechanical engineers.

To improve the usability of the environments for configuration space exploration for operators, it would be useful to characterize the reduced configuration spaces for given jobs. In particular, when multi-objective optimization is relevant

for objectives such as maximizing throughput and minimizing waste, it would be useful if CSX could indicate the tradeoff between these objectives.

Acknowledgment. We thank the reviewers for their feedback. 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. We thank Bas Hermus for providing a 3D drawing of perfect binding. This work is related to the European patent application EP3855304 A1 which is published on 28 July 2021.

References

1. van Antwerpen, H., Néron, P., Tolmach, A.P., Visser, E., Wachsmuth, G.: A constraint language for static semantic analysis based on scope graphs. In: Erwig, M., Rompf, T. (eds.) Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 49–60. ACM (2016). https://doi.org/10.1145/2847538.2847543
2. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 01.7. A language and toolset for program transformation. Sci. Comput. Program. **72**(1–2), 52–70 (2008). https://doi.org/10.1016/j.scico.2007.11.003
3. Dageförde, J.C., Kuchen, H.: A compiler and virtual machine for constraint-logic object-oriented programming with muli. J. Comput. Lang. **53**, 63–78 (2019). https://doi.org/10.1016/j.cola.2019.05.001
4. van Deursen, A., Klint, P., Tip, F.: Origin tracking. J. Symb. Comput. **15**(5/6), 523–545 (1993)
5. Erdweg, S., et al.: Evaluating and comparing language workbenches: existing results and benchmarks for the future. Comput. Lang. Syst. Struct. **44**, 24–47 (2015). https://doi.org/10.1016/j.cl.2015.08.007
6. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002). https://doi.org/10.1145/505145.505149
7. Kats, L.C.L., Visser, E.: The Spoofax language workbench: rules for declarative specification of languages and IDEs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, pp. 444–463. ACM, Reno/Tahoe (2010). https://doi.org/10.1145/1869459.1869497
8. Keshishzadeh, S., Mooij, A.J., Mousavi, M.R.: Early fault detection in DSLs using SMT solving and automated debugging. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 182–196. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40561-7_13
9. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74970-7_38
10. Neron, P., Tolmach, A., Visser, E., Wachsmuth, G.: A theory of name resolution. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 205–231. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_9
11. de Souza Amorim, L.E., Visser, E.: Multi-purpose syntax definition with SDF3. In: de Boer, F., Cerone, A. (eds.) SEFM 2020. LNCS, vol. 12310, pp. 1–23. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58768-0_1

12. Stoel, J., van der Storm, T., Vinju, J.J.: AlleAlle: bounded relational model finding with unbounded data. In: Masuhara, H., 0001, T.P. (eds.) Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, 23–24 October 2019, pp. 46–61. ACM (2019). https://doi.org/10.1145/3359591.3359726
13. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008–2013. AI Mag. **35**(2), 55–60 (2014)
14. Torlak, E., Bodík, R.: Growing solver-aided languages with rosette. In: Hosking, A.L., Eugster, P.T., Hirschfeld, R. (eds.) ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH 2013, Indianapolis, IN, USA, 26–31 October 2013, pp. 135–152. ACM (2013). https://doi.org/10.1145/2509578.2509586
15. Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_49
16. Voelter, M.: The design, evolution, and use of KernelF. In: Rensink, A., Sánchez Cuadrado, J. (eds.) ICMT 2018. LNCS, vol. 10888, pp. 3–55. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93317-7_1
17. Voelter, M., Koščejev, S., Riedel, M., Deitsch, A., Hinkelmann, A.: A domain-specific language for payroll calculations: an experience report from DATEV. In: Domain-Specific Languages in Practice, pp. 93–130. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-73758-0_4