Navigating Through Digital Printing Systems

The Use of a Domain-Specific Language for Route Finding in Digital Printing Systems

Bram van Walraven

Navigating Through Digital Printing Systems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Bram van Walraven
born in Amsterdam, The Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: The author generated this image in part with DALL-E 2, OpenAI's large-scale image-generation model. Upon generating the draft image, the author reviewed, edited, and revised the image to their own liking and takes ultimate responsibility for the content of this publication.

The writer was enabled by Canon Production Printing Netherlands B.V. to perform research that partly forms the basis for this thesis. Canon Production Printing Netherlands B.V. does not accept responsibility for the accuracy of the data, opinions and conclusions mentioned in this thesis, which are entirely for the account of the writer.

Navigating Through Digital Printing Systems

Author: Bram van Walraven
Student id: 4340507

Abstract

Digital printing systems allow for the production of a large variety of different products. Making production plans for all these different products is challenging. One of the challenging aspects of making these production plans is choosing the right sequence of machines, to produce the desired intent. This is challenging due to three aspects: the large number of interdependent variables in the problem instances, the variability of machines, and the search for the best solution from a large set of valid solutions. In this thesis, we implement and evaluate the use of a domain-specific language (DSL) called RSX (Routing Space eXploration), to assist in choosing a sequence of machines. We do this together with an industrial partner. For RSX we use a model-driven approach, and it can be used to model the devices, production steps, and product properties of the digital printing domain. It transforms those into a constraint model described in the MiniZinc language, which is used as input for a constraint solver. We present the implementation of the RSX language and MiniZinc constraint model, and we evaluate the language coverage, accuracy, and performance. From these evaluations, we conclude that RSX can be used to model a number of cases, which were characteristic in the context of our industrial partner. Furthermore, we conclude that RSX can compile and solve the evaluated cases in the order of a few seconds and that the implementation is accurate, such that it can be used as a proof of concept.

Thesis Committee:

Chair:	Prof. dr. A. E. Zaidman	Faculty EEMCS, TU Delft
Committee Member:	Dr. B. P. Ahrens	Faculty EEMCS, TU Delft
Committee Member:	M. Brunner, MSc.	Canon Production Printing
Committee Member:	J. Denkers, MSc.	Faculty EEMCS, TU Delft

Thesis Advisor:	Prof. dr. E. Visser	Faculty EEMCS, TU Delft

Preface

It has been a year since I started working on this thesis. Half a year earlier, the preparations for this master thesis project had already started. The reason for this timely planning was absolutely not the result of my extensive efforts in finding a thesis project. While I was counting the days to the summer holidays, Eelco Visser asked me what my interests were for my master thesis. Startled by this question, I had to admit that it had not even occurred to me to think about that. Luckily, Eelco seized this opportunity to promote all the interesting projects at his research group. Due to the inspiring lectures by Eelco during previous courses and my general interest in programming languages, I was easily convinced. The project at Canon Production Printing seemed to be the right fit, and before I knew it, I was on my way to Venlo for my first day.

It saddens me that this project did not end the same way it started: with Eelco as my supervisor. It is thanks to all the other people involved in this project, that I was able to finish it. From the start of my project, Marvin Brunner and I had daily, online meetings in which we discussed the progress of my project. I find it incredible that he could bring up the patience with my cluelessness in how to tackle this project. Jasper Denkers, while being busy with his PhD, also guided me in the right direction, especially in getting my research on paper. The feedback from Louis van Gool, being more of an outsider to my particular project, was essential as well. I want to thank these people for being invested in this project from start to end.

Andy Zaidman and Benedikt Ahrens were involved a few months after the project started, but without hesitation, they decided to take over the supervision of my project. While it might not be a project within their main research focus, they still took the time to help me and that is something I am very grateful for.

Looking back on this year, I am happy that Eelco was a bit more decisive than I was in getting my project started. I have learned many things. Not only about programming languages or research in general, but also about writing, planning and communication. All essential skills for future endeavors.

Bram van Walraven
Rotterdam, The Netherlands
November 2, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The printing industry finds its roots in ancient times. Early civilizations used clay tablets and papyrus rolls to write down information [28]. Nowadays, we would not see this process of manual writing as part of the printing industry anymore. However, this was one of the earliest forms of information sharing by other means than oral communication. It serves the same goal as the printing industry today, which is the creation of physical matter to share knowledge between people. Luckily, we do not have to rely on clay tablets and papyrus rolls anymore and have much more advanced techniques available in our modern times. Today, we can print thousands of books, newspapers and any other printed matter in only a few hours or less.

This advancement in the printing industry is owed to the invention of the printing press and later, digital printers. Even though the invention of these advanced machines makes printing easier than before, there are still issues which operators of these machines need to solve manually. The most difficult problems to solve currently are in the development of a production plan for these printed products. One of these problems is that the operator needs to know which devices to use to make the printed product and in which order to use these devices. This is what we call the *routing problem*. It is not straightforward to solve and often has to be performed manually by an operator. Automation techniques assisting in solving this problem could potentially help the operators in solving this problem. In this thesis, we describe a case study into the use of a domain-specific language (DSL) with a constraint modelling language as a backend to automatically generate solutions to the problem and therefore assisting in overcoming the challenges of the routing problem in digital printing systems.

In this chapter, we will discuss the background of digital printers and, by extension, digital printing systems in general. Furthermore, we will outline the context of our research, and we will give an overview of the contributions of this thesis.

1.1 Digital Printing Systems

Digital printing systems are manufacturing systems which can flexibly produce different forms of printed matter. This includes, but is certainly not limited to, books, calendars, magazines, textile, cardboard and wallpaper. A digital printer is the primary component of such digital printing systems. Alongside this printer, an array of devices, called finishers, are used to combine the printed material into a finished product. In this section, we will describe these two types of devices.

1.1.1 Digital Printers

The main component of a digital printing system is a digital printer. These printers can use a different number of techniques to recreate a digitally loaded image onto almost any surface. Household printers and larger office printers are examples of digital printers that almost everyone is familiar with. While this technique is used by many for their daily printing needs, this technique competes with a much older way of printing. Mainly in the process of printing books, offset printing is a common technique. Offset printing finds its origins in the Middle Ages and makes use of printing plates to press the content of a page onto a sheet of paper. While the making of printing plates was a tedious manual process before, modern systems can make them automatically. Although the automation of this process reduces the costs of making these printing plates, this is still an expensive operation. The advantage is that this only needs to happen once for each exact copy of the book and can then be reused for any amount of copies.

The older technique of offset printing is very straightforward and optimized due to centuries of development. However, this technique requires extensive preparation, due to the printing plates which need to be made for each different book and, if applicable, for each different version of a book. While the costs of actually producing larger quantities of books using offset printing are low compared to digital printing, it is less profitable to make printing plates for lower quantities.

Digital printers do not suffer from this problem. A digital printer can print different books or even different kinds of printed matter, without creating and changing printing plates. This makes digital printers interesting for producing products which are published in lower volumes. Furthermore, it makes extensive customization of the printing more viable. An example of this is personalized photo albums, which are unique for the person that orders the photo album. It is also not surprising that this technique is used for on-demand book printing. In on-demand printing, books are not printed in advance and then stored in warehouses. Only after a customer explicitly orders a book online, the book is printed. This can greatly reduce costs and reduces the risk of overproducing.

1.1.2 Finishers

Digital printers are not the only equipment that is required to produce printed matter in its finished form. Other machines, so-called finishers, play a crucial role in the production printing process. They execute all the steps required to get from a collection of printed sheets to a final product. These machines can for example ensure that a stack of printed sheets is turned into an actual book. The steps required depend on the type of product that is produced. Examples of product types are ring-bound books or perfect-bound books, but also folded flyers or birthday calendars. These product types might have the same set of printed sheets and are produced by the same printer, but they can require different finishing steps. For a ring-bound book, there needs to be a step in the production process that punches holes into the sheets and one which binds the stack of punched sheets together using a ring wire. For a perfect-bound book, the stack needs to be milled and glued first, before a cover sheet is glued to the book block. These finishing steps are mostly done by different machines, often produced by different manufacturers.

Apart from the flexibility of digital printers themselves, these different types of finishers also introduce a form of freedom in digital printing systems. Not only does the individual configuration of each specific type of finisher allow for more options, the choice of which finisher to use is also a new form of flexibility that has been introduced in digital printing systems. The introduction of this flexibility comes with the introduction of a complex problem: choosing the right finisher for the job. This problem needs to be solved before a product can be produced.

1.2 Routing in Digital Printing Systems

In this thesis, we will discuss the routing problem in digital printing systems. This problem will be covered extensively in Chapter 2. Routing in digital printing systems occurs whenever there are two or more steps in the production process that need to be executed. Then, a decision must be made in which order to execute these steps. Depending on the type of printer and finishers, there can be a lot of freedom in this routing and thus many possible routes. Some printers and finishers are configured to be *inline*. This means that together they form one big machine, in which the product automatically moves from one machine to the other. The advantage of this configuration is that it requires less manual work to make a product. However, some flexibility is lost, because the steps can only be executed in one specific order. Other printers and finishers are configured to be *offline*. This means that they can be seen as completely separate machines and that they do not automatically move any intermediate product from one machine to the other. This has as advantage that doing so gives a lot of flexibility in constructing the production line. The disadvantage is that it also requires more manual work to ensure the product is placed in the right machine, at the right moment.

Both approaches and also combinations of these approaches are used in printing shops, i.e. shops that produce printed products. It depends on the main goal of a printing shop whether they use inline or offline equipment. If they want to provide flexibility in the types of products they offer, they would most likely use offline equipment. If they want to provide a more efficient and faster way of producing, they would rather go for inline equipment.

1.3 Contributions

In this thesis, we perform a case study about the implementation and application of a domain-specific language (DSL) which tries to help reduce the challenges an operator faces when trying to solve the routing problem. We call this language RSX (Routing Space eXploration). Before we explain the implementation of RSX, we first present a more detailed overview of the routing problem and the challenges of this problem. It is important to understand these difficulties because in our case study, we try to minimize these as much as possible by implementing RSX.

RSX compiles to MiniZinc [20]. MiniZinc is a constraint programming language, which integrates with multiple constraint solvers. We can use such a solver to solve the routing problem for us, instead of developing an algorithm ourselves. Unfortunately, we do not get the solution to our problem for free. We get back the challenge of having to model our problem in a constraint language. We present a way of modelling our problem directly in MiniZinc, which is then used as a basis for the compilation of RSX.

For the implementation of RSX, we used the Spoofax Language Workbench [17]. In this language workbench, we define the syntax and static semantics specifications. We also use this workbench for defining transformations of RSX to MiniZinc. In this thesis, we present the exact implementation of these aspects of the language.

We evaluated the language on three different aspects. First, we evaluated whether the language covers the domain enough to define realistic cases in RSX. To achieve this we performed multiple think-aloud co-design sessions together with domain experts. Secondly, we evaluated whether the language is correct and complete by implementing a test suite for the language. Last, we evaluated the language performance by benchmarking the compilation and solving times for different cases developed in RSX.

To summarize, we make the following contributions.

- We design an approach of modelling the routing problem in a constraint model (Chapter 3).

- We design and implement a language, RSX, which allows to model a digital printing setup, which can be used to find a production route (Chapter 4).
- We evaluate the coverage, accuracy and performance of RSX (Chapter 5).

1.4 Research Context

The research for this thesis is performed in collaboration with our industrial partner Canon Production Printing. Canon Production Printing is a company developing digital printers. Every printer comes with control software, which is used to control both the printer and its finishers. When these finishers are placed inline, the control software directly manages the finisher. Using this software, a production plan can be created which is executed automatically by the printer and finishers.

Making the production plan using this control software requires considerable knowledge and is mostly done by experienced operators. To aid in this process, Canon Production Printing has been trying to encode the knowledge of these operators into a system that can help with the decision-making in building a production plan. The language CSX was developed as research into configuration space exploration [9]. By abstracting over a constraint model, CSX helps an operator in finding a valid or optimized parameter configuration for an intent.

In its current form, CSX has one limitation, which is that the sequence of actions performed is fixed. However, products can have multiple valid production routes. These different routing options cannot be explored with CSX alone. Therefore, in this thesis, we do a case study into the development of another language named RSX. The focus of this language is on researching the possibility to also help with choosing a correct sequence of actions.

Chapter 2

The Routing Problem

When you ask people to describe a book, flyer or calendar they want to print, they are often very capable of doing so, albeit up to a limited level of detail. It is easy to describe the content, the cover page or the type of cover for a book. However, when asking people if they could describe the steps required to make their desired book, they will encounter more difficulty. Most of the time, the reason is that people lack knowledge of bookmaking. That is not so surprising, but even when people would have a basic understanding of bookmaking, this is a difficult question. Bookmaking, or production printing in general, is not just a single physical action; it involves many, different aspects. The printing process starts with a series of questions like how to make the book block, the cover page or the binding. Each question refers to a small step in the production process. The operator of the machines that are going to make the product, can therefore use the answers to these questions to come up with a production plan.

Such a production plan is easily created when the choices of intent are limited. If all books look the same, an operator could just executes the same production plan over and over. However, as you can see when looking at your bookshelf, each book is different. Not only do they differ in their content, but the way they look is also different. They have different sizes, paper, covers and bindings. Furthermore, the variety of printed applications reaches much further than just books. Digital printers can print, for example, banners, cardboard, textile and wallpaper as well. This large variety in applications is the reason that developing production plans is more involved than choosing a predefined plan.

In this chapter, we will discuss how finding a production route is a problem that needs to be solved. We discuss why this problem is difficult to solve and what potential solutions there are to solve the problem, based on the difficulties discussed.

2.1 The Routing Problem Explained

To fully understand the challenges of the routing problem which we try to solve using RSX, we will first explain the problem in more detail.

2.1.1 An example

To explain what the routing problem entails, we will use an example. Imagine that we are an operator at a printing shop. Our task is to produce the orders that the printing shop gets from its customers. Because we know our customers have very different needs and wishes about what they want their books to look like, we have a large set of different machines at our disposal to be able to produce almost any kind of book.

Now, what happens when we receive an order from a customer? Imagine our customer is a student that wants to print their thesis as a small book. We need to know what the

student wants their book to look like, and they, in turn, need to know what their options are. Therefore, we discuss the possibilities with them. We ask questions like: what kind of binding do they want? what type of cover sheet? what should the size of the pages be? Using their answers, we try to come up with an intent specification. This specification should specify the aspects of the book and describe the book in such a way that the student will be happy with the book that we produce.

Now that we know what the book needs to look like, we can determine how we are going to produce it. We start by looking at the machines we have at our disposal and ask ourselves: which machines do I need to use to get the book that the student wants? Say that the student has decided on the perfect-bound binding method. From our knowledge as a trained operator, we know which actions we need to perform. A perfect-bound book is not easy to make, so we will describe the actions in random order below.

- ***Gluing*** A perfect-bound book is a book where the cover is glued to the book block. This is different from other binding methods where the book block is bound using a wire or by stitching. Thus, a logical action to perform is that the glue needs to be applied to the book block.
- ***Milling*** Because we want a high-quality perfect binding, we need to ensure that the glue sticks well to the book block. This is done by a process called milling. In milling, one edge of the book block is made rough, so the glue adheres better to the block.
- ***Covering*** The cover needs to be glued to the book block. This means there needs to be an action which applies the cover to the book block. Usually, this is done by folding a cover sheet over the book block.
- ***Creasing*** To ensure that the book is easy to read, and the cover sheet wraps nicely around the book block, the cover sheet is creased. This means that small creases are applied to the cover sheet in the places where it folds around the book block and where the cover needs to open up.
- ***Printing*** A very logical step in the process is printing. A thesis without pages would not result in a high grade. So, somewhere in our production route, we need to make sure that the sheets are printed.
- ***Gathering*** Printing happens on a sheet-by-sheet basis. To ensure we have a book block which we can cover with a cover sheet, we need to gather all the sheets together to form a book block.
- ***Trimming*** When gathering sheets and during the covering of the book block, it is very likely that the sheets are not perfectly aligned. Therefore, trimming the non-bounded edges can make sure that the book has clean edges without any pages or the cover sticking out.

Now that we know which actions to use, we ask ourselves: in what order do we need to execute these actions? For this, we take a look at what we know about each action. Again, from our knowledge of being a trained operator, we know that when applying the glue to the book block, we need to have a milled edge first. Therefore, we know that somewhere before gluing, we need to do milling. We also know that an edge needs to be glued before applying the cover sheet. So, before covering, we need to do gluing. An overview of the requirements for each action is given in Table 2.1. Based on this knowledge, we try to come up with an order in which we need to execute the actions. It is important that for the order, all requirements for the actions are met. An example of a valid route can be seen in Figure 2.2. This route is valid because the requirements for the actions are met. For example, as discussed before,

Action	Requires	Produces
Gluing	milled book block	glued book block
Milling	book block	milled book block
Covering	glued book block + creased sheet	perfect-bound book
Creasing	sheet	creased sheet
Printing	sheet	printed sheet
Gathering	sheets	book block
Trimming	book block	trimmed book block

Table 2.1: Requirements and products for all actions needed to make a perfect-bound book.

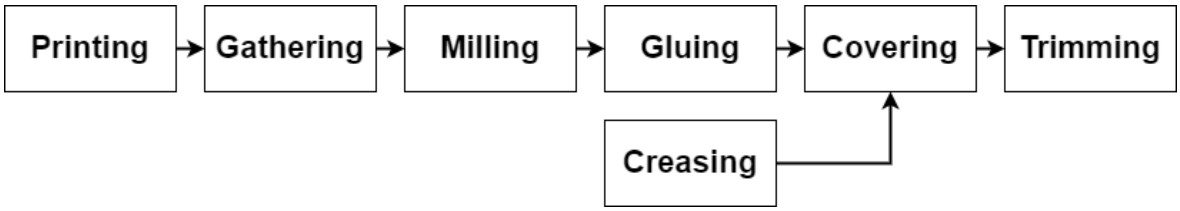

Figure 2.2: A possible order of actions to produce a perfect-bound book.

gluing needs to happen after milling and milling after gathering. If we swapped the milling and gathering actions, we would get an invalid route.

Note that this route is not a simple sequence of steps, but involves the combination of two parts of the route in the covering step. The reason for that is that when executing the covering action, you need both a cover sheet and a book block. These require different actions and therefore have different routes before being joined in the covering action.

So, are we ready to make the book? Almost, but there is still one question that remains. How do we configure the machines that execute these actions? The student wants their perfect-bound thesis in a specific size. So we need to configure the machines to ensure that the book will be that size. There are two actions during which the size of the final product changes. During milling, a part of one edge will be milled off and during trimming a part of the book block will be trimmed off. To ensure that the book block is of the specified size, we need to configure the milling and trimming machines in just the right way so that the final book block is the correct size. This is dependent on the size of the input sheets, so before we are ready to produce the book, we need to make sure these parameters are configured correctly.

After all these questions have been answered, we have a complete production plan. We know which actions and machines to use, in which order we need to execute these actions and what the configuration parameters are. Using this production plan we can start producing the book and, if everything goes to plan, the machines can do the rest. Note that for this example we have formalized the process of coming to a production plan extensively. In real life, most of the steps given in this example are not made in the exact order we present them here. Furthermore, the specification is rarely completely clear, and an operator might make assumptions about what the customer wants. The operator can also often choose from different actions that achieve the same result, making the decisions even more complex.

2.1.2 Routing Problem Definition

To generalize the example given in the previous section, the three aspects that an operator encounters when trying to come up with a production plan are the following.

1. The operator needs to know which actions are needed in the production plan.
2. The operator needs to know in which order to execute these actions.

3. The operator needs to know how to configure the parameters of those actions.

The first two questions can be seen as part of the routing problem. The answers to these questions result in the route a product makes during the printing process. However, these aspects of coming to a production plan are not independent of one another. As you might expect, the question of how to configure an action is dependent on when the action is executed in the sequence of actions that form the production process. Also, the question of when to execute an action is dependent on which actions are chosen in the first place. These dependencies are also present in the opposite direction. Thus, the point in the production process where an action is chosen, is dependent on the configuration parameters of the action and the actions that are chosen is dependent on the order of those actions. There might be configuration parameters that influence whether a certain order of actions is valid. Furthermore, whether a certain position of an action in a route is decided can influence which other actions can subsequently be used. This makes the problem more difficult, and we will go deeper into why it is difficult in Section 2.2.

Our focus will be on the first two aspects stated in this section: which actions need to be executed and in which part of the process? The third aspect is already tackled in earlier research [9]. When needing to decide on those questions, an operator needs to oversee the whole process. This becomes problematic when the different properties of the intended product and the number of steps to get to the intent increase. When the intent is only to have a single printed sheet of a certain size, it is quite easy to find a production route that produces this sheet. However, when someone wishes to have a high-quality bounded book of a few hundred pages, this becomes a lot more challenging. Even more so, when there is a need of doing so efficiently.

The problem that arises, is that the operator needs to know if an action can be executed before another. There are situations where the order of actions is very strict and where it is very clear which action should come first. For example, when forming the book block, an operator wants to ensure that the pages of the block are in the correct order and are printed. So when forming the book block, e.g. by making a stack of sheets, it is required that the sheets are printed beforehand. In this case, the printing step needs to be done before gathering. It is difficult to do this the other way around because printing a stack of sheets can only be done by taking the stack apart into individual sheets again. An operator needs to be aware of these constraints when making a production plan.

However, the consequences of some actions might not be immediately clear and only noticed after the action is executed. An example of this is when gathering sheets, the machine executing this action might not be able to nicely align all the sheets in such a way that the stack does not contain slightly skewed sheets. In general, this is perceived as ugly and something that needs to be prevented. This is usually done by trimming the edges of the complete stack so all the sheets get the same size. However, to trim the edges, the sheets need to be larger than their desired size, as trimming will always reduce the size of the sheets. Thus, to ensure that sheets still get the right size and are not skewed, it is important to take into account that these steps need to be in this order and that the sheets that are stacked are initially larger than the desired sheet size.

Furthermore, another interesting aspect that arises from the earlier example where an operator prints sheets first, before stacking them, is that there is physically nothing preventing the operator from stacking sheets first, taking the stack apart, printing the sheets, and then stacking them again. However, it is clear that this involves more steps and is therefore less efficient. Additionally, the more you separate the sheets, the higher the chance that the sheets will get damaged. This shows what occurs more often in the domain, namely that there are a lot of possibilities in the routing, but that some of these possibilities are neither logical nor desirable. Another example is that when trimming a stack to its desired size, this can be done by trimming each edge once, by trimming only two edges, or by multiple com-

binations of these two. Again, there is nothing physically preventing the operator to choose one of these routes, but they still need to make a decision on which of these options to use.

To summarize the routing problem, we can give the following formal definition of the problem:

> The routing problem is the problem that arises whenever an operator of digital printing systems needs to choose a set of actions to execute and needs to choose in which order to execute them. He needs to choose these in such a way that the product satisfies the intent, and such that each action can be executed at its location in the route, without any unwanted side effects.

2.2 The Difficulty of the Routing Problem

In the previous section, we discussed what the routing problem entails. It is difficult for an operator to find a route that adheres to the constraints of individual actions and still produces the intended product. The operator needs to choose the right actions, at the right position in the route to ensure the desired result. To make this easier for an operator, special software could be used to assist the operator. This way the operator would not require the extensive domain knowledge currently demanded from them. It would be very helpful if there was software that could point out to the operator whether a chosen route is valid or better yet, could suggest a valid route for the specified intent. Such software currently does not exist and a good reason for that could be that building such software is not trivial. This section will explain the three reasons why solving the routing problem is difficult, even when delegating this task to a computer.

2.2.1 Complexity of Solution Space

The first reason is that the problem is very complex. To show this, we can make use of an analogy between cooking and the production of books. Say that you had dinner with a friend. At the dinner, your friend made an amazing soup for you. After you compliment them, they tell you that they just followed the recipe, and it was not much of an effort. Encouraged by these words and because you enjoyed the soup so much, you want to make the soup yourself. You do not have the recipe, but you remember what it looked like and what it tasted like. Is that enough to make the soup yourself? Maybe if you were a very experienced cook and the soup is simple, it would be doable, but most people would have trouble recreating the soup. Your friend had a recipe that they could follow, in which the ingredients and steps to make the soup are described. However, for you, it is very difficult to guess which ingredients they used, which steps they took, and in which order they put the ingredients together.

This problem is analogous to the routing problem in the production printing domain. To show this, we will take a look again at the example given in Section 2.1.1. There we tried to find a production route that produces the thesis of a student as a perfect-bound book. In our analogy the soup can be compared to the thesis from our earlier example. It is the product we try to create. The actions and the order of actions are analogous to the steps of the recipe for the soup. Now, imagine that we look at the routing problem the other way around. Then we would have a production route and the question becomes: what would we be able to produce by that route? Then the problem would be much simpler. If we know that in our production route we mill, glue, and then cover, we can easily see that we get a perfect-bound book. Just as when following a recipe to make a dish, we can follow the steps to see what it produces. However, when reasoning from the point of intent, this is not a problem in which we simply follow all the actions anymore. Then we also need to guess which steps we need to do in which order and which "ingredients" we need to use to get our desired intent.

The examples in this analogy are both instances of satisfiability problems. In these problems, the goal is to determine whether a mathematical formula is satisfiable [4]. In general, satisfiability problems belong to the class of problems which cannot be solved in polynomial time. This means that the solving time of the problem could grow exponentially with the size of the problem. The larger the problem instance, the more difficult it becomes to find a satisfactory solution to this formula. Because in the routing problem this mathematical formula consists of many variables, which are also interdependent, an algorithm that can solve the problem efficiently is impossible to find. The difficulty here is therefore to find an algorithm that can find a solution in a still acceptable amount of time, by optimizing the specific instances of satisfiability problems we are dealing with.

2.2.2 Variability of Production Lines

The problem described in the previous section could be surmountable by extensively optimizing the algorithm on the possible solutions. This could be done, for instance, by reducing the solution space by removing any wrong solutions. However, the variability in the intent is not the only thing that needs to be taken into consideration. Although this is indeed the only problem an operator faces, the developer of an algorithm that solves the routing problem also faces variability in production lines. While an operator has only a limited set of machines at their disposal, a developer of an algorithm should solve for all possible combinations which are used in the field. In our previous example of printing the thesis, we explained that the operator had a lot of different machines at their disposal. However, some printing shops have a more limited set of machines available. Furthermore, printing shops could have different versions of the same machines, with different limitations. In the real world, printing and finishing systems consist of tens of different finisher types, which are produced by different manufacturers. Operators have different subsets of finishers available. Furthermore, there is a continuous development of new finishers with different capabilities.

This makes the routing problem more difficult to solve. As said earlier, we could find an algorithm that is optimized for a very specific set of possible actions, but when these sets are different for each operator, there would be limited use for the algorithm. Furthermore, the algorithm would only be useful for a limited amount of time, because any change to the machines would make the algorithm unusable. The challenge that is introduced here is that we need to find an algorithm that can be optimized for different sets of available actions which could change over time.

2.2.3 Searchability of Solution Space

The third challenge lies in finding the right solution to the problem. In some situations, there is an almost endless set of solutions that result in the same product. To give a more concrete example, say that the student of our previous example wants to have their thesis in a small book of 148mm by 210mm, which is the size of an A5 sheet. The problem is that we do not have this exact paper size available and that we need to trim the book after we glued the cover sheet to it. Therefore, we need to have a larger sheet size for our book block than intended before trimming. The question of how much bigger is dependent on how much we trim. However, nothing is preventing us from trimming twice, thrice, or even more times. Say that we need to trim 20mm from one edge to get to the intended size of 148mm. Then we could trim 20mm in one go. We could also trim 10mm twice. We could also trim 5mm once and 15mm once. As you can see, there are a lot of different possibilities, which all produce our valid intended size.

To help the operator in finding a production route, an algorithm needs to choose one of these configurations. However, to do so, the algorithm needs to know what the value of a certain configuration is. Some configurations might be faster, but also result in more

paper loss. Other configurations might reduce the paper waste, but take longer. Often, it is dependent on the situation which of these configurations is desirable. The challenge that is introduced here is that an algorithm needs to decide which configuration to give to an operator to properly assist the operator. This is what we call the searchability of the solution space. If the solutions that we get are all valid, which one do we have to choose?

2.3 Potential Solutions for the Routing Problem

Analyzing and describing the problem can be interesting from a researcher's point of view, but in the end, our goal is to solve the problem in the best way possible. Therefore, we also present potential solutions. The combination of the three difficulties described in the previous section gives us an indication of what the three requirements are for a potential solution to the problem. To ensure that the solution is helpful for an operator, it should address these three challenges. Therefore, the potential solution needs to be able to solve a large problem instance, be able to be easily extendable, and be able to navigate the solution space for an instance of the problem. In this section, we will take a look at some techniques that could aid in solving the routing problem. These techniques fall into one of two categories: a retrieval-based approach or a generative approach.

2.3.1 Retrieval-based solutions

A retrieval-based solution to the routing problem would be to predefine the possible routes that an operator can choose from. This might sound like an inadequate solution. However, if we had a printing shop that only produces one single type of book, this is the perfect solution. We can experiment with different production routes and optimize our production line for this specific intent. Once we have done so, there is no need to create any other routes. Even if we would slightly increase the intent options we offer, this option is still viable. We can manually devise specific solutions for each customization we want to offer. The only problem is that at some point there is a limit to the customization that you can offer. It is impossible to manually devise all possible solutions. Furthermore, it becomes problematic whenever there is a change in the production line. Whenever a machine is replaced by another or a new machine is added to the production line, the fixed routes have to be adjusted to accommodate this change. Therefore, this solution would limit the utilization of all the different capabilities a machine has to offer. The potential of these machines then goes to waste, only because there is no set of all fixed routes available for these machines.

2.3.2 Generative Solutions

A generative solution to the routing problem is different from a retrieval-based solution. Instead of choosing a known solution, we generate the solution based on the specific problem instance. This has the advantage that we can potentially solve all possible problem instances and not only the problem instances we devised beforehand. However, the question then becomes how we are going to find this solution. In this chapter, we see that it is not easy to find such a solution. Two approaches can potentially help in finding a solution.

Data-Heavy Solutions

Data-heavy solutions could be used to let a computer learn what a viable production route is. By using a lot of data about successfully executed production routes, a computer could gain the same knowledge as an experienced operator. This could be a viable option, as long as there is enough data available to train the model. Without enough data, the model could give routes that do not produce the intended result or routes of which an operator says that

it is not reasonable to execute it in that way. Both problems can become a nuisance because it would mean that either an operator needs to check every single production plan generated by the model or there is a risk that products are produced which do not abide by the intent. Currently, there is not a lot of data available about production plans and their intent. Therefore, within our domain, this is not a viable solution.

Rule-Based Solutions

While there is a limited set of data available on production plans, there is a lot of knowledge available about the domain. This knowledge is taught in special education programs for operators in the printing industry. Therefore, an idea is to model this knowledge into an algorithm using constraint modeling. A rule-based solution allows us to encode this knowledge into an expert system. We could describe the limitations of our machines and let the system itself find a solution that abides by these limitations. This is a viable solution as long as we can realistically model our knowledge into these rules. However, that can be a challenge, which is also recognized by other authors [14, 13].

Chapter 3

Solving the Routing Problem: A Constraint Model

To find a solution to the problem described in the previous chapter, we make use of the method of constraint programming. In our research, we use the programming language MiniZinc [3]. MiniZinc is a declarative language for constraint programming. This means that the programmer defines a set of variables and constraints. These variables and constraints are compiled so that a generic solver can be used to find a value for these variables, which satisfies the constraints. This way, if you can describe your problem as variables and constraints, that is the only thing you have to do. It is not required to also describe an algorithm for finding a solution to the problem. A solver takes care of that, by applying a smart brute-force search on your solution space. The task of a programmer in MiniZinc is thus to correctly describe the problem, in such a way that the solver finds a correct solution. In this chapter, we start by giving a motivation for the use of MiniZinc. Then we describe an implementation of a MiniZinc model that describes the routing problem in digital printing systems. After that, we will present a summary of the constraint model and how a solver finds solutions to the routing problem.

3.1 Motivation for MiniZinc

RSX is compiled into the MiniZinc constraint programming language. However, more languages can be used to model constraint problems. Before we chose MiniZinc as the target language, we looked at a few other languages and made a decision for MiniZinc. This decision is described in this section.

3.1.1 Criteria

The target language is the language that RSX is translated to. The goal of the target language is, therefore, to be able to define the constraints that a satisfactory solution for the routing problem adheres to. Computing a solution is not something that is expressed in the target language and there is no necessary one-to-one relationship between a language and a solver. However, it is still important to take the solver into account when choosing a target language. The target language is useless without a solver that can find a viable solution.

Based on this constraint-solving approach, we defined a few criteria to help in choosing a target language. These were divided into two parts. The *must-haves* and the *nice-to-haves*. The must-haves are the criteria that the language should abide by without exceptions. If a language does not follow all of these criteria, the language is not a valid choice for our target language. The nice-to-haves are the criteria that are not a necessity, but of which it would be good to have as many as possible in the target language. These criteria make it easier to

model our problem in the target language. The must-haves for the language are described below.

Must-haves

1. In the language it must be possible to declaratively define the problem as described in the previous chapter.
    a) The language should support defining boolean and integer constraints.
    b) The language should support defining constraints over collections of integers or enumerable types.
2. There should be a solver for the language that can find a solution to the problem described in the language.
3. The solver should be able to find an optimal solution based on an optimization function.

The first criterion describes that in the language, it should be possible to declaratively define the problem using constraints. *Declaratively* here means that only the constraints themselves should be expressed in the language itself and not the way that a solution to the problem can be found. This is in contrast to a language that *imperatively* describes the way that a solution can be found. The two types of constraints are based on the problem that needs to be solved. These are the minimum type of constraints expected, at this moment, to be needed to describe the problem.

The second and third criteria describe the need for a solver that can be used in combination with the target language. There needs to be a solver that can find a solution based on the constraints and also order multiple solutions based on a function to decide on the best solution. Without this last part, any solution is seen as equal in value to the solver and there is no guarantee that the solver finds the best solution.

Now that we have defined the minimum criteria for our target language, we will take a look at the criteria which describe language and solver features that would be nice to have in our target language. These are described below.

Nice-to-haves

1. The solver can solve the problems which are present in the domain in time and space efficient way.
2. The language supports abstractions like functions to model repetitive tasks.
3. The solver can show the possible values for variables based on the constraints given.
4. The solver can show the constraints enforcing values for variables based on the constraints given.
5. The language is well documented and several examples of programs in the language exist.
6. The language describes the constraints in such a way that it is easy for a human to understand what the described constraints intend to achieve.

The first item on the list describes the need to have a solver that can efficiently solve the problem. In theory, a solution can be found by trying all possible combinations of parameters until the result satisfies the constraints. However, this is not a time-efficient way of solving the problem given numerous parameters and would therefore most likely not suffice. Therefore, a solver should be smart enough to give a solution faster than a brute-force algorithm.

The second item concerns the efficiency of describing the problem in the language. To ensure that repetitive constraints do not have to be described multiple times, abstractions like functions or predicates are a nice-to-have feature in the modeling language.

The third and fourth items describe the nice-to-have query capabilities of the solution space. There are cases in which the objective is not necessarily to find a solution that satisfies the constraints, but more so to describe the possible values a variable can have. Another aspect one might be interested in, are the constraints that limit the value of a certain variable. This way, it is possible to see why a solver came to a given solution.

The fifth item states that it is desirable to use a language that is well documented and for which multiple examples exist. The reason is that without good documentation, modeling the problem in such a language becomes unnecessarily difficult. This is not desirable for a research project which is not focussed specifically on finding the best modeling language for the problem.

The last item describes the wish for the target language to be easily interpreted by a human reader. The reason for this is that, especially during development, the intent of the constraints described should easily be understood by other parties involved in the project. Again, not having a human-readable language makes modeling unnecessarily difficult.

3.1.2 Choice

Based on the criteria described in the previous section, we need to make a choice on a language to model the problem in. For a language to be considered, the language should at least follow the must-have criteria. Based on the nice-to-haves, a decision can be made for one of these languages. Three languages are the main candidates for the target language. These are:

1. MiniZinc [20]
2. IDP-Z3 [6]
3. Prolog, with the `clpfd` library [8]

These three languages implement all the must-have criteria, although Prolog only when using a specific library. The advantages and disadvantages of each language, based on the nice-to-have criteria, are discussed below.

MiniZinc

MiniZinc is a widely used constraint programming language built on top of FlatZinc. This is a language that is developed as a *solver-input* language. It is implemented for several solvers that each have different specializations. MiniZinc makes it possible to define constraints over integers, boolean and floating-point numbers and to reason about these using universal and existential quantifiers or using logical connectives. The ability to use different solvers allows for choosing a solver that is best for the problem at hand. However, it is mainly focused on finding a solution, and it is more difficult to explore the solution space by querying the possible values for a specific variable.

IDP-Z3

IDP-Z3 is a language that is developed at the KU in Leuven. Just as with MiniZinc, it allows defining constraints over integers, boolean and floating-point numbers. It is also possible to reason about those, similarly to in MiniZinc. It only supports the Z3 solver and is, therefore, more limited with regard to choosing the best solver for a specific problem. However, it does allow for extensive querying of the solution space. It can decide which values are allowed

for certain variables and by which constraints they are constrained. An example of this can be found in the Interactive Consultant, which is built on top of IDP-Z3.

Prolog

Prolog is another widely used language, but not necessarily in the field of constraint programming. When using the general language, it is not possible to reason about constraints in Prolog, due to the requirements it enforces on the instantiation of unknown variables. Therefore, a library was developed to solve this problem. Using this library, it is possible to define constraints in Prolog, in a way that is similar to MiniZinc or IDP-Z3. However, this is not as natural as in the other two languages, as Prolog was never developed to be used as a constraint programming language.

Final Choice

Based on the discussion about these three languages, we decided to use MiniZinc as the target language for modeling the routing problem. The language is widely used and well-documented. This makes it a more attractive choice over the IDP-Z3 language, which - although it has almost all the same language features - is much more of an academic research language. It is therefore less well documented than MiniZinc, which is widely used in the industry. Furthermore, MiniZinc is written to define constraint programs and has an easier syntax and semantics for defining such problems. There is only one disadvantage of using this language and that is the fact that it is more difficult to explore the solution space for a problem described in MiniZinc. However, this seems to be a problem that is mainly on the solver side and not on the modeling side. Since for a proof-of-concept implementation the focus will be more on the modeling side, this should not impose a big problem in this phase of the research.

3.2 Constraint Model Implementation

In this section, we give an implementation of our problem in a constraint model. We use an example to show how the different aspects of the problem can be modeled in MiniZinc. The example consists of a small setup. We have two actions: `ToTrim` and `ToOrientate`. The first action trims the top edge of a stack and the other action rotates the stack by 90 degrees in a clockwise direction.

3.2.1 Route

The ultimate goal in our problem is to find a route. In Chapter 2 we described this route as a sequence of actions. For our case study, we assume that a route can always be described by a *linear* sequence of actions. This excludes routes that contain different branches. The reason we make this assumption is that this simplifies our problem.

In MiniZinc, we describe this sequence as an array of variables. Each variable in the array describes an action that is taken at a step in the sequence. The first variable describes the action taken in the first step, the second variable the action taken in the second step, and so on. The array is described in MiniZinc using the following definition.

```
1 par int: maxActions = 4;
2 enum ACTION = {ToTrim, ToOrientate, ToSkip};
```

```
3 array[1..maxActions] of var ACTION: actions;
4 constraint actions[maxActions] = ToSkip;
```

Each variable in the array is of type `ACTION`, which is a type that describes all the possible actions in our domain. Here we have the two actions of our example and an extra action `ToSkip`. This last action is to model the fact that we can also do nothing in a step. This is required, because arrays in MiniZinc have a fixed length. When would have a route that only requires three steps, we still need to define an action for the fourth position in our array.

In the actions array, each value describes the action to take in each step of our sequence. The length of the array is the maximum number of actions we can perform and thus the maximum length of our sequence. A solver for MiniZinc can now automatically find a value for each variable in the array. Because we have not restricted the actions using constraints in MiniZinc, the solver can return any possible action for each variable. Thus, to get a useful solution from the solver, we need to extend our model first.

3.2.2 Product

In our domain, a sequence of actions produces a product. Each action modifies the product in some way and the sum of all these changes forms the final product. Ultimately we want to reason about the aspects of this product, thus we need to find a way to model a product in MiniZinc. One way to describe a product is by a set of properties that specify the aspects of the product we want to model. In MiniZinc, we describe these properties as an array of variables, where each variable in a property array describes the state of that property at a certain step in the sequence. The following MiniZinc code gives the modeling of the properties in our example.

```
1 array[1..maxActions] of var bool: isTopTrimmed;
2 array[1..maxActions] of var bool: isRightTrimmed;
3 array[1..maxActions] of var bool: isLeftTrimmed;
4 array[1..maxActions] of var bool: isBottomTrimmed;
5
6 constraint isTopTrimmed[1] = false;
7 constraint isRightTrimmed[1] = false;
8 constraint isBottomTrimmed[1] = false;
9 constraint isLeftTrimmed[1] = false;
```

Here, we describe the array of variables for four properties. Each property models whether a certain edge is trimmed or not. The type of the properties here is a boolean but can be any MiniZinc type. Again, the length of the array is equal to the maximum number of actions in our sequence. We also add a constraint for each property which defines the initial value for that property. For our example, we assume that each edge is not yet trimmed at the start.

3.2.3 Postcondition

In the previous section, we described a way to model the product as a set of properties. Each action that we choose to perform, applies changes to the product. These changes are something we need to model as well. In MiniZinc, the changes of an action can be modeled by a constraint over the properties of the product after the execution of an action. In the following code snippet, we show how we model this for the actions and properties in our example.

```
constraint forall(i in 1..maxActions-1)(actions[i] == ToTrim -> isTopTrimmed[i+1]
↪  == true);
constraint forall(i in 1..maxActions-1)(actions[i] == ToTrim ->
↪  isRightTrimmed[i+1] == isRightTrimmed[i]);
constraint forall(i in 1..maxActions-1)(actions[i] == ToTrim
↪  ->isBottomTrimmed[i+1] == isBottomTrimmed[i]);
constraint forall(i in 1..maxActions-1)(actions[i] == ToTrim ->
↪  isLeftTrimmed[i+1] == isLeftTrimmed[i]);

constraint forall(i in 1..maxActions-1)(actions[i] == ToOrientate ->
↪  isTopTrimmed[i+1] == isLeftTrimmed[i]);
constraint forall(i in 1..maxActions-1)(actions[i] == ToOrientate ->
↪  isRightTrimmed[i+1] == isTopTrimmed[i]);
constraint forall(i in 1..maxActions-1)(actions[i] == ToOrientate ->
↪  isBottomTrimmed[i+1] == isRightTrimmed[i]);
constraint forall(i in 1..maxActions-1)(actions[i] == ToOrientate ->
↪  isLeftTrimmed[i+1] == isBottomTrimmed[i]);

constraint forall(i in 1..maxActions-1)(actions[i] == ToSkip -> isTopTrimmed[i+1]
↪  == isTopTrimmed[i]);
constraint forall(i in 1..maxActions-1)(actions[i] == ToSkip ->
↪  isRightTrimmed[i+1] == isRightTrimmed[i]);
constraint forall(i in 1..maxActions-1)(actions[i] == ToSkip ->
↪  isBottomTrimmed[i+1] == isBottomTrimmed[i]);
constraint forall(i in 1..maxActions-1)(actions[i] == ToSkip ->
↪  isLeftTrimmed[i+1] == isLeftTrimmed[i]);
```

We define the following constraint for each action: if the action is chosen, the changes that are applied to the property must hold in the step after the action was chosen. For example, we say that if the `ToTrim` action is chosen, we set the value of the `isTopTrimmed` property to `true` in the following step. The other properties stay the same. This models the fact that the `ToTrim` action only trims the top edge of our stack. The `ToSkip` action models an action where we do not do anything. Therefore, we say that the properties keep their value in the step after the `ToSkip` action.

3.2.4 Precondition

Some actions have preconditions on the product that the action is applied on. This means that an action can only be executed if the properties of the product it changes, satisfy these preconditions. This can be modeled in MiniZinc by adding constraints on the properties of the product at the moment an action is executed. We can add a precondition to our example in the following way.

```
constraint forall(i in 1..maxActions-1)(actions[i] == ToTrim -> isTopTrimmed[i]
↪  == false);
```

Here, the `ToTrim` action can only trim the top edge, if it is not already trimmed. We model this by adding a constraint on the value of the `isTopTrimmed` property at the step where the `ToTrim` action is chosen.

3.2.5 Intent

In our problem, we want the product to satisfy the intent. Because we model the product as a set of properties, this set of properties should satisfy the intent as well. To describe whether the properties satisfy the intent, we should look at the value of the property after we applied all actions in our route. In our MiniZinc implementation, this is described by the last value in the array that describes the property. To ensure that this value satisfies the intent, we can add a constraint to our MiniZinc model. For example, we add the following constraints.

```
constraint isTopTrimmed[maxActions] = false;
constraint isRightTrimmed[maxActions] = false;
constraint isBottomTrimmed[maxActions] = true;
constraint isLeftTrimmed[maxActions] = false;
```

We add a constraint for each of our four properties. The constraint says that the last value in the array which describes the properties should satisfy the intent. In this case, the intent is that only the bottom edge is trimmed. The other edges are explicitly not trimmed. Note that it is not required to describe the intent for all properties. Then, the property becomes *free*, which means that we do not care what the value is, and we accept any value the solver comes up with. For example, if we would not define whether the top edge should be trimmed, the solver can both give a solution in which the top edge is trimmed and one in which it is not.

3.2.6 Action Order

Not all actions can be chosen freely in the domain. Sometimes, we can only apply an action if we first apply another action. We can describe this in MiniZinc using a constraint that says: if an action is chosen, then the other action must be chosen in the next step of the sequence. An example of our two actions is shown below.

```
constraint forall(i in 1..maxActions)(actions[i] == ToOrientate <-> actions[i+1]
↪  == ToTrim);
```

Here the action `ToOrientate` is always followed by the action `ToTrim`. Because we use a bi-implication, the action `ToTrim`, can only be used in a step, if the action `ToOrientate` is used in the previous step.

3.3 Summary

Together, the translations of domain concepts to MiniZinc, as described in the previous sections, form a core constraint model for our routing problem. To summarize, the solver tries to find a solution to all variables we have defined and ensures that for each of the values it finds, the constraints hold. In our model, the most important variables are described by the action array. The values that the solver finds for these variables describe the actions we need to take. Because we also add the variables for properties, the constraints over the actions, and

the constraints over the intent to the model, we ensure that the solver only chooses actions that are valid in our domain.

So, what happens if we run the solver on our MiniZinc model? For this example, we leave out the action order constraint described in Section 3.2.6. The solver comes up with the following solution.

```
1 actions = [ToTrim, ToOrientate, ToOrientate, ToSkip, ToSkip];
2 isTopTrimmed = [false, true, false, false, false];
3 isRightTrimmed = [false, false, true, false, false];
4 isLeftTrimmed = [false, false, false, false, false];
5 isBottomTrimmed = [false, false, false, true, true];
```

The route that we follow is described by the array of actions. We first need to trim the stack, then rotate it twice, and then do nothing. As we see by the values in the property arrays, after executing these actions, the bottom of our stack is trimmed and the other edges are not. This is exactly what we described in our intent in Section 3.2.5.

It can easily be verified that the above solution is valid. However, there are more solutions that are valid. We can ask our solver to give us all possible solutions. For this specific problem instance, there are five possible solutions. One of the other options is given below.

```
1 actions = [ToOrientate, ToTrim, ToOrientate, ToOrientate, ToSkip];
2 isTopTrimmed = [false, false, true, false, false];
3 isRightTrimmed = [false, false, false, true, false];
4 isLeftTrimmed = [false, false, false, false, false];
5 isBottomTrimmed = [false, false, false, false, true];
```

In this solution, we orientate the stack first before we trim. Because all the edges are untrimmed at the start of our route, the stack after a rotation is equivalent to the stack before the orientation. It might seem obvious that the first orientation action here is unnecessary. However, this example shows that the constraint solver is only as smart as the model we feed it. If we do not explicitly define the fact that the first rotation is unnecessary, the solver cannot deduce that from our model.

Chapter 4

Solving the Routing Problem: RSX

In the previous chapter, we showed a way of modeling the routing problem using constraints. Using a generic constraint solver, we can find a solution to the problem instance. As we have seen in that chapter, it was quite straightforward to model the problem instance in a constraint model. However, if we would extend this model, the number of variables and constraints will grow significantly. Then, choosing the right predicates and keeping an overview of the available actions can become more difficult. Furthermore, changing production lines can be challenging because it requires the problem definition to change over time. To aid in this modeling, we propose a domain-specific language (DSL) called RSX. RSX allows giving a high-level definition of the actions and the properties an operator has at their disposal. This high-level description is translated into the MiniZinc language. The translation generates a constraint model based on the model described in Chapter 3. This constraint model can then be used to find a production route. In this chapter, we will go into more detail about why we chose to build a DSL and what this DSL looks like.

4.1 Why a Domain-Specific Language?

The first question that might come time mind is: why do we need a DSL for this purpose? Why do we even invest in developing a DSL? Constraint modeling is a widely researched topic, and it is not surprising that several solutions already exist in helping someone model their problem as constraints. MiniZinc, which we have seen in Chapter 3, is one of the tools that can help us to define a constraint model. However, using only a constraint language has some disadvantages for our problem.

One disadvantage of constraint modeling is the difficulty of expressing your intent. Note that we mean the intent of the programmer of the constraint model here, not the intent in our routing problem. There are many ways of describing the problem in constraints and without expertise, it is difficult to formulate a good model [14]. Furthermore, at our industrial partner, constraint modeling is not a widely used technology. Asking programmers of control software to define a model in low-level constraints, is therefore not something that they have experience with.

Moreover, in our problem, we need more than a few models to be defined in a constraint language. One of the challenges of the problem is the variability of the devices we have to model. If we only had to model a few of these devices, it would probably be acceptable to try and overcome the challenges of programming directly in a constraint language. However, because of the large number of devices, it becomes more difficult.

For that reason, an investment in the development of a DSL is more interesting. A DSL promises that it is easier to understand for a user with knowledge of the domain. This makes it easier to express intent for problems in that domain. However, building and maintaining a

DSL is not free and it requires considerable work to ensure that these promises can be lived up to.

Whether the advantages of a DSL outweigh the investment needed for our specific problem, is something we cannot say at this moment. In our case study, we study the use of a DSL, so the result of our research could help in answering that question.

4.2 Language Development

We developed RSX in the language workbench Spoofax [17]. Spoofax and language workbenches in general strive to allow for the implementing, testing, and deploying of programming languages, using meta-languages for aspects like syntax in SDF3 [23]; semantics in Statix [2]; and transformation definitions in Stratego [5]. By combining these important aspects of language design into one IDE, the goal of a language workbench is to increase the productivity of the language designer [12].

Inspired by the development of other DSLs in Spoofax [25, 9], we decided to develop our language incrementally and iteratively. This means that the phases of analysis, design, and implementation of the language are combined into small iterative steps, which are repeated in the process of language development. This process is supported by the features and meta-languages which are implemented in Spoofax. Small iterations are possible because Spoofax provides us a functioning IDE and testing environment, without the need of implementing the complete language. This makes an agile approach to language development possible and natural.

4.3 An Example in RSX

To give an idea of what the language looks like, we will start with a small example in which we model a production line. This example has the goal of showing the syntax, semantics, and how an RSX program is transformed into a MiniZinc model. In the other sections of this chapter, we will go deeper into these concepts, and we will give more formal specifications of those.

Our example consists of the modeling of a problem that an operator sometimes faces in a real printing shop. As an operator, we have the intent to have a trimmed stack of sheets. In the storage room, we already have a stack, but none of the edges of this stack is trimmed. The only two devices we have available in our print shop are a device that can perform the action of rotating a stack 90 degrees in a clockwise direction and a device that performs the action of trimming the top edge of the input stack, rotating it 180 degrees, and then trimming the top edge again. Using these two devices, we are going to try to produce our intended trimmed stack. How we do that is something our RSX model is will help us with.

4.3.1 Properties

We start our model by defining our properties. For each edge of the stack, we are defining a boolean property that says whether that edge is trimmed or not. Figure 4.1 shows how these properties are defined in our RSX program. As you can see, we have four lines describing each property. These are defined by the `property` keyword, a name, and a type.

4.3.2 Devices and Actions

Now that we have defined our properties, we can define our devices and actions. As mentioned, we only have two devices available in our printing shop, each with a single possible action. We can model these directly in our RSX model. Next to these two devices, we add

```
property isTopTrimmed: bool
property isRightTrimmed: bool
property isBottomTrimmed: bool
property isLeftTrimmed: bool
```

Figure 4.1: Properties of the example RSX model.

an extra device that models the action of us fetching a stack from our storage. Later, in Section 4.6.6 we will go deeper into the reason why we require this extra action. How these actions are modeled is shown in Figure 4.2.

All devices are defined by their name after the `device` keyword. A device can have multiple paths. The `Trimmer` device here has two paths and the other two devices only have a single path. Each of these paths gives an order of actions that can be followed in the device. This means that when an action in a path is used in a route, the rest of the actions in that path need to be performed as well. The `Trimmer` for exapmle has a path that performs the `ToTrim` action, the `ToOrientate` action twice and then the `ToTrim` action again. This to simulate a trimmer that always trims the top and bottom edges of a stack.

The paths can also have a length of zero actions. This means that the device can be bypassed, i.e. the device is used, but does not perform any actions. In the example, the `Trimmer` device has such a path. Such an empty path can be useful when a device is *inline*. The `Trimmer` and `Rotator` devices are inline, which means that these devices are physically connected and that the `Rotator` cannot be used without using the `Trimmer`. This is modeled using the `inline` keyword, with a path of devices that are inline.

Each action defined in a device should also be defined by the `action` keyword, a name, and a set of constraints between curly brackets. These constraints give the conditions which must hold when one of these actions is chosen at a certain point in the route. A constraint is given by the `constraint` keyword and an expression that evaluates to a boolean value. In these constraints, the properties are referred to by their name, as defined by their property definition, and an `in` or `out` keyword. This keyword describes whether we are reasoning about the property value before or after the action. A property with an `in` keyword describes the value of the property before a step where the action is chosen and out describes the value of the property after a step where the action is chosen. We can use these values to reason about the pre- and postconditions of the action.

As explained earlier, the `ToFetch` action describes the action of taking a stack from our storage room. When we take a stack, the stack is not trimmed. Therefore, we say that when the `ToFetch` action is chosen, all the edges are not trimmed afterward. Thus, the constraints are added for the properties that the value is false after the action is executed. Because before we fetch a stack from the storage room, there is no stack we can use, we say that the trim properties are absent. This models the fact that we cannot reason about these properties when there is no stack.

The `ToTrim` action can only trim the top edge of the input stack. So, when we do a `ToTrim` action, the result is that the top edge is trimmed. The `ToTrim` action, therefore, has the constraint that the top edge is trimmed after the action is executed.

In the `ToOrientate` action, we rotate the stack 90 degrees in a clockwise direction. Therefore, when the top edge is trimmed, the right edge becomes trimmed. When the top edge is not trimmed, the right edge will not be trimmed after the orientation action. The same holds for the other edges. This is expressed by the four constraints which say that the value for the properties after the action is executed is equal to the value of the property of the edge it was rotated from.

```
inline Trimmer -> Rotator

device StorageRoom {
  path ToFetch
}

device Trimmer {
  path ToTrim -> ToOrientate -> ToOrientate -> ToTrim
  path
}

device Rotator {
  path ToOrientate
}

action ToFetch {
  constraint isTopTrimmed.in.absent and isTopTrimmed.out == false
  constraint isRightTrimmed.in.absent and isRightTrimmed.out == false
  constraint isBottomTrimmed.in.absent and isBottomTrimmed.out == false
  constraint isLeftTrimmed.in.absent and isLeftTrimmed.out == false
}

action ToTrim {
  constraint isTopTrimmed.out == true
}

action ToOrientate {
  constraint isRightTrimmed.out == isTopTrimmed.in
  constraint isBottomTrimmed.out == isRightTrimmed.in
  constraint isLeftTrimmed.out == isBottomTrimmed.in
  constraint isTopTrimmed.out == isLeftTrimmed.in
}
```

Figure 4.2: Devices and actions, including their constraints of the example RSX model.

Intent

The last thing we need to specify is the intent. Without intent, our constraint model does not know what we want and therefore thinks that any route is valid. For our intent we want all edges to be trimmed. The specification of this intent in RSX is shown in Figure 4.3. There, you can see that the intent is declared by the intent keyword and an expression evaluating a boolean value. For each edge, we specify the property, and whether it is trimmed, to be true. We use the `in` keyword to define that this is a precondition on the intent. It does not make much sense to define a postcondition on the intent, thus the `out` keyword cannot be used in the intent specification.

4.3.3 Generation of MiniZinc

Now that we have defined a model for our problem, we can use this model to find a solution to our problem. Our RSX compiler transforms the model into MiniZinc variables and con-

```
intent isTopTrimmed.in == true
intent isRightTrimmed.in == true
intent isBottomTrimmed.in == true
intent isLeftTrimmed.in == true
```

Figure 4.3: Intent specification of the example RSX model.

straints. When we run the compiler we get the MiniZinc code as shown in Figure 4.4. Later in this chapter, in Section 4.6, we will go deeper into how each part of the RSX model gets translated into MiniZinc.

```
enum UNIT = {present};
int : maxActions = 12;
int : intMinValue = -intMaxValue;
int : intMaxValue = 10000;
array[1..maxActions] of var ACTION : actions;
constraint actions[maxActions] == ToSkip;

constraint forall(i in 1..maxActions - 1)(actions[i] == ToSkip -> actions[i + 1]
↪  == ToSkip);
enum ACTION = {ToSkip, StorageRoom_ToFetch_0, Trimmer_ToTrim_0,
↪  Trimmer_ToOrientate_0, Trimmer_ToOrientate_1, Trimmer_ToTrim_1,
↪  Trimmer_ToBypass_0, Rotator_ToOrientate_0};
constraint forall(i in 1..maxActions - 1)(actions[i] == ToSkip -> isTopTrimmed[i]
↪  == isTopTrimmed[i + 1]);
constraint forall(i in 1..maxActions - 1)(actions[i] == ToSkip ->
↪  isRightTrimmed[i] == isRightTrimmed[i + 1]);
constraint forall(i in 1..maxActions - 1)(actions[i] == ToSkip ->
↪  isBottomTrimmed[i] == isBottomTrimmed[i + 1]);
constraint forall(i in 1..maxActions - 1)(actions[i] == ToSkip ->
↪  isLeftTrimmed[i] == isLeftTrimmed[i + 1]);

array[1..maxActions] of var opt bool : isTopTrimmed;
constraint isTopTrimmed[1] == <>;
array[1..maxActions] of var opt bool : isRightTrimmed;
constraint isRightTrimmed[1] == <>;
array[1..maxActions] of var opt bool : isBottomTrimmed;
constraint isBottomTrimmed[1] == <>;
array[1..maxActions] of var opt bool : isLeftTrimmed;
constraint isLeftTrimmed[1] == <>;

constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_1 \/
↪  actions[i] == Trimmer_ToBypass_0 <-> actions[i + 1] ==
↪  Rotator_ToOrientate_0);
constraint not (actions[1] == Rotator_ToOrientate_0);
constraint forall(i in 1..maxActions - 1)(actions[i] == StorageRoom_ToFetch_0 ->
↪  absent(isTopTrimmed[i]) /\ isTopTrimmed[i + 1] == false);
```

```
27 constraint forall(i in 1..maxActions - 1)(actions[i] == StorageRoom_ToFetch_0 ->
   ↪ absent(isRightTrimmed[i]) /\ isRightTrimmed[i + 1] == false);
28 constraint forall(i in 1..maxActions - 1)(actions[i] == StorageRoom_ToFetch_0 ->
   ↪ absent(isBottomTrimmed[i]) /\ isBottomTrimmed[i + 1] == false);
29 constraint forall(i in 1..maxActions - 1)(actions[i] == StorageRoom_ToFetch_0 ->
   ↪ absent(isLeftTrimmed[i]) /\ isLeftTrimmed[i + 1] == false);
30
31 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_0 <->
   ↪ actions[i + 1] == Trimmer_ToOrientate_0);
32
33 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_0 <->
   ↪ actions[i + 1] == Trimmer_ToOrientate_1);
34 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_1 <->
   ↪ actions[i + 1] == Trimmer_ToTrim_1);
35 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_0 ->
   ↪ isTopTrimmed[i + 1] == true);
36 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_0 ->
   ↪ isRightTrimmed[i] == isRightTrimmed[i + 1]);
37 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_0 ->
   ↪ isBottomTrimmed[i] == isBottomTrimmed[i + 1]);
38 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_0 ->
   ↪ isLeftTrimmed[i] == isLeftTrimmed[i + 1]);
39 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_0 ->
   ↪ isRightTrimmed[i + 1] == isTopTrimmed[i]);
40 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_0 ->
   ↪ isBottomTrimmed[i + 1] == isRightTrimmed[i]);
41 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_0 ->
   ↪ isLeftTrimmed[i + 1] == isBottomTrimmed[i]);
42 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_0 ->
   ↪ isTopTrimmed[i + 1] == isLeftTrimmed[i]);
43 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_1 ->
   ↪ isRightTrimmed[i + 1] == isTopTrimmed[i]);
44 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_1 ->
   ↪ isBottomTrimmed[i + 1] == isRightTrimmed[i]);
45 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_1 ->
   ↪ isLeftTrimmed[i + 1] == isBottomTrimmed[i]);
46 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToOrientate_1 ->
   ↪ isTopTrimmed[i + 1] == isLeftTrimmed[i]);
47 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_1 ->
   ↪ isTopTrimmed[i + 1] == true);
48 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_1 ->
   ↪ isRightTrimmed[i] == isRightTrimmed[i + 1]);
49 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_1 ->
   ↪ isBottomTrimmed[i] == isBottomTrimmed[i + 1]);
50 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToTrim_1 ->
   ↪ isLeftTrimmed[i] == isLeftTrimmed[i + 1]);
51 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToBypass_0 ->
   ↪ isTopTrimmed[i] == isTopTrimmed[i + 1]);
52 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToBypass_0 ->
   ↪ isRightTrimmed[i] == isRightTrimmed[i + 1]);
53 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToBypass_0 ->
   ↪ isBottomTrimmed[i] == isBottomTrimmed[i + 1]);
```

```
54 constraint forall(i in 1..maxActions - 1)(actions[i] == Trimmer_ToBypass_0 ->
   ↪ isLeftTrimmed[i] == isLeftTrimmed[i + 1]);
55
56 constraint forall(i in 1..maxActions - 1)(actions[i] == Rotator_ToOrientate_0 ->
   ↪ isRightTrimmed[i + 1] == isTopTrimmed[i]);
57 constraint forall(i in 1..maxActions - 1)(actions[i] == Rotator_ToOrientate_0 ->
   ↪ isBottomTrimmed[i + 1] == isRightTrimmed[i]);
58 constraint forall(i in 1..maxActions - 1)(actions[i] == Rotator_ToOrientate_0 ->
   ↪ isLeftTrimmed[i + 1] == isBottomTrimmed[i]);
59 constraint forall(i in 1..maxActions - 1)(actions[i] == Rotator_ToOrientate_0 ->
   ↪ isTopTrimmed[i + 1] == isLeftTrimmed[i]);
60
61 constraint forall(i in maxActions..maxActions)(isTopTrimmed[i] == true);
62 constraint forall(i in maxActions..maxActions)(isRightTrimmed[i] == true);
63 constraint forall(i in maxActions..maxActions)(isBottomTrimmed[i] == true);
64 constraint forall(i in maxActions..maxActions)(isLeftTrimmed[i] == true);
```

Figure 4.4: Generated MiniZinc code for the example RSX model.

Now that we have transformed our model into a MiniZinc model, we can use a solver to solve the constraint problem. We can choose what solution we are interested in. It could be that we just want any solution, but we could also want to optimize a certain variable or a function of multiple variables. For this example, we try to find any route. We do this by manually adding a statement in the last line that states that we just want a satisfactory solution to our problem. When we now run this MiniZinc program with the Gecode solver [1], we get the result as shown in Figure 4.5.

```
1 actions = [StorageRoom_ToFetch_0, Trimmer_ToTrim_0, Trimmer_ToOrientate_0,
  ↪ Trimmer_ToOrientate_1, Trimmer_ToTrim_1, Rotator_ToOrientate_0,
  ↪ Trimmer_ToTrim_0, Trimmer_ToOrientate_0, Trimmer_ToOrientate_1,
  ↪ Trimmer_ToTrim_1, Rotator_ToOrientate_0, ToSkip];
2 isTopTrimmed = [<>, false, true, false, false, true, false, true, true, false,
  ↪ true, true];
3 isRightTrimmed = [<>, false, false, true, false, false, true, true, true, true,
  ↪ true, true];
4 isBottomTrimmed = [<>, false, false, false, true, true, false, false, true, true,
  ↪ true, true];
5 isLeftTrimmed = [<>, false, false, false, false, false, true, true, false, true,
  ↪ true, true];
```

Figure 4.5: Solution of the Gecode solver for the example RSX model.

The solution we get from our solver is only a single solution. For this specific problem, there are multiple valid solutions. In this particular solution, we see that all the variables in our MiniZinc model have an assigned value. In our case, this is the array of actions and the four properties we have defined in our RSX specification. The array of actions gives us the route of actions we need to execute. Each action is prefixed by its device and suffixed by a unique identifier. This way, we can differentiate between actions with the same name used in different devices and at different paths within devices.

[1] https://www.gecode.org/

We start our route by fetching the stack from our storage room. Then, we trim the stack, rotate it twice, trim again, rotate it again, trim it again, rotate it twice more, and then trim it one last time. The rotation action which is used as the last action is not required for our intent. However, the action is required because the `Rotator` device is inline with the `Trimmer`. Therefore, we always need to perform a `ToOrientate` action after we use our `Trimmer`. We also see that at the last position in all the property arrays, the value of the property is true. This is also what we had specified and if we execute these actions in real life, we would indeed get a trimmed stack.

4.4 Syntax

Now that we have defined and shown an example of RSX, we will go deeper into the different parts of designing RSX. The syntax of RSX is designed in such a way that it requires as little knowledge about the language as possible to understand an RSX model. This means that the design tries to make the syntax explicit. We want the reader of the RSX code to immediately understand the concepts encoded in the model. All definitions in the RSX model are, therefore, clearly described by a keyword from which it should immediately be clear what the definition describes. For example, properties in the language are described by the `property` keyword, and constraints are described by the `constraint` keyword.

The reason for this explicit syntax is the fact that the language is to be used in a very specific domain. The language tries to be as close as possible to this domain, to ensure that someone with a lot of domain knowledge can easily understand the language. If the language syntax would not be explicit, it would negate the effects of the language being close to the domain.

Another part of the syntax of RSX where this reasoning is used is the representation of some logical expressions. While most expressions in RSX are inspired by their MiniZinc equivalent, the syntax for some expressions intentionally deviates. The `\/` and `/\` expressions in MiniZinc represent an *or* and *and* expression respectively. This notation is common in predicate logic and probably for that reason chosen in MiniZinc. However, outside this domain and specifically in the domain of digital printing systems, this notation is not used as much. Therefore, in RSX, these expressions are denoted using the `or` and `and` keywords, which make it immediately clear what they mean, even for someone without in-depth knowledge of constraint programming.

The complete grammar of the language is described in Figure B.1. In this grammar, *ID* is used to denote an identifier in the language. These identifiers need to match the regular expression `[a-zA-Z\][_a-zA-Z0-9]*`. All identifiers matching this expression are valid, except for identifiers which match keywords in the language: `action`, `unit`, `int`, `nat`, `float`, `bool`, `true`, `false`, `property`, `preserve`, `constraint`, `not`, `intent`, `device`, `path` or `exit`. Integer values are represented by *i*. Any value matching the regular expression `[0-9]+` is a valid integer value. Negations in the language are done using a specific negation expression.

4.5 Static Semantics

RSX is a statically typed language. This means that the types of expressions are known at the time of compilation. The choice for a statically typed language instead of a dynamically typed language is because MiniZinc is statically typed. Therefore, during compilation, the types of expressions must be known to be able to generate the correct MiniZinc code. Furthermore, a static type system is also useful for RSX itself. It is practical to have clear types for properties, to better understand the domain concept they describe.

To ensure that an RSX model can always be transformed into a correct MiniZinc program, a set of typing rules was defined. These typing rules ensure that every expression in RSX

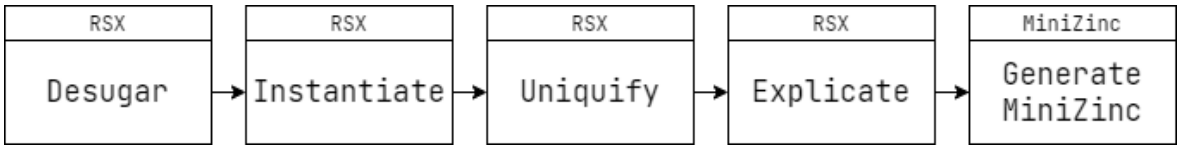

Figure 4.6: Compiler passes for the RSX compiler. Above each pass, the language of the generated code in that pass is shown.

has an equivalent correct expression in MiniZinc. The exact typing rules for the language are shown in Appendix C. The actual type checking of RSX was implemented in Statix [2].

4.6 Transformation

The language concepts that can be described in the RSX language need to be translated into MiniZinc code by the RSX compiler. To achieve this, we combine a set of rewrite strategies with a set of rewrite or transformation rules. These strategies and rules are defined in Stratego [5]. This language is designed for program transformations and is part of the Spoofax Language Workbench. In this section, we will give a general overview of how the program transformation from RSX to MiniZinc works.

4.6.1 Transformation Structure

The transformation rules are the rules that define each construct in RSX, and how they should be transformed into MiniZinc constructs. During the implementation of the transformation rules for our RSX compiler, the focus was on writing transformations that were as simple as possible. The performance of the compiler was less of a priority. The reason is that it was expected that the language would change significantly during the development and evaluation sessions as described in Chapter 5. Making the transformations simple and therefore easy to follow, makes it easier to make changes to these transformations. To make these transformations as simple as possible, a structure was designed based on the *nanopass* compiler architecture [22]. In this architecture, the compiler uses multiple smaller passes over the program, which each performs one single small task. The idea behind this is that each step or stage can be easily tested and that extensions to the source language can be easily implemented.

For RSX, this meant that during development we could easily extend the language by adding an extra pass to the existing set of passes, with minimal changes to the existing passes. The passes that exist in RSX are shown in Figure 4.6. In total there are five different passes. The first four passes transform the RSX model into another RSX model. These passes change the model or add information to the model to make it more straightforward to generate MiniZinc code in the last pass of the compiler. The five stages of the compiler each have a specific task, namely:

1. **Desugar** Removing syntactic sugar from the language.
2. **Instantiate** Generate instances for actions used in devices.
3. **Uniquify** Make names in the language unique.
4. **Explicate** Add explicit definitions to the language.
5. **Generate MiniZinc** Transform the RSX code into MiniZinc.

To give more details about each of these compiler passes, we will describe each stage in the following sections in more detail.

4.6.2 Desugar

In this first stage of the RSX compiler, syntactic sugar is removed from the language. Syntactic sugar is the syntax that is equivalent to other syntax existing in the language and is mostly added to make writing these constructs easier. In RSX, this is only used for defining actions without parameters. Instead of having to write parentheses with no parameters in between, it is also possible to leave out these parentheses. An example of this difference can be seen in Figure 4.7. There, it is shown that the abstract syntax tree (AST) of the two actions is different, even though these two actions are equivalent. An abstract syntax tree is a representation of the program which shows the structure of the program in the form of a tree. In the desugaring stage, the first action will be transformed into the form of the second action. This way, in the rest of the compiler pipeline, there is no need to define transformations for the action construct without parameters.

```
action ToExample1 {}
action ToExample2() {}
```

(a) Example of the difference between an action without parameters and an action with zero parameters in RSX.

```
RsxModel(
  [
    Action("ToExample1", []),
    ActionWithParams("ToExample2", [], [])
  ]
)
```

(b) Example of the difference between the AST of an action without parameters and the AST of an action with zero parameters in RSX.

Figure 4.7: Example of the difference between an action without parameters and an action with zero parameters in RSX and their respective ASTs.

4.6.3 Instantiate

The second stage of the compiler has the task to instantiate the actions in a device. Each action that is used on a path in a device, can be seen as an instance of that action. It is the implementation of that action in a device. Instead of directly transforming the action definitions to MiniZinc, the instantiations of an action will be translated to MiniZinc.

An instance of an action is the same as the action definition, but with a unique name based on the device it is used in. This way, later in the transformation process, it is possible to differentiate between the different instances of an action. Figure 4.8 shows an example of the instantiation of an action in two devices. The `ToInstantiate` action is used in two devices, in one of which it is even used twice. This means that there are three instances of this action. Therefore, in the instantiated version of RSX, there are three definitions of the `ToInstantiate` action. Each of these has a unique name, which is based on the device the instance is used in. Note that the original action definition is removed from the instantiated version of RSX. This means that when an action definition is never used in a device and therefore never instantiated, this action will also not be translated to MiniZinc. An example

is the `ToNeverUse` action. This action is not used in one of the two devices and therefore not present anymore in the instantiated version of RSX.

The empty path in a device gets instantiated as well. This empty path is used to denote a bypass of the device. To instantiate this specific bypass, an extra action is instantiated. This bypass action does not have any constraints or parameters and therefore does not change any properties. Therefore, whenever this bypass action is chosen, it is as if the device was bypassed. An example of such a bypass action can be seen in Figure 4.8.

4.6.4 Uniquify

In the uniquify step, we ensure that all identifiers in the language are unique. This is only important for action parameters. Because these are scoped by the action they are used in, different parameters can have the same name in two different actions. However, we do not have an action scope in the generated MiniZinc code and must therefore ensure that these names do not overlap. In Figure 4.9 an example of the uniquify step is shown. The two defined actions both use a parameter `unique`. In the uniquified RSX model, these parameters have the name of the action added to them, to ensure that these parameter names are unique.

4.6.5 Explicate

The fourth pass of the compiler is the explicate stage. This pass makes implicit language constructs explicit in the language. The RSX model always contains a `ToSkip` action. This action is required to make sure that a route can have fewer actions than the maximum number of possible actions. This action does not need to be defined for every RSX model but is automatically added to the model in the explicate pass.

Another implicit construct is that when a property is not constrained on its output value in an action, its value should be preserved. This means that the value should be the same after the execution of the action, as it was before the action. To achieve this in RSX, a preserve construct is automatically added to an action if it does not reason about an output value of a property. Figure 4.10 shows an example of the RSX explicate pass. As can be seen there, the `ToSkip` action is automatically added. Also, the output value of the property `preserved` is not constrained in the `ToExplicate` action. Therefore, a preserve for this property is added to the action.

4.6.6 Generate MiniZinc

The last stage of the compiler generates the final MiniZinc code. Because of all the earlier small steps, it is quite straightforward to transform the RSX code to MiniZinc. Every top-level definition of the RSX model can directly be translated into MiniZinc. Together with a prelude, these translated definitions form the whole generated MiniZinc code. The following sections explain how each of these definitions and the prelude are transformed and generated.

Prelude

The MiniZinc prelude is a piece of MiniZinc code that gets added to the generated MiniZinc code, regardless of the RSX model. It defines common values and types that could be used in the generated MiniZinc code. The prelude is shown in Figure 4.11. It first defines the unit type. This type can only have one value and is represented by a MiniZinc enum with just a single value. This value is the enum value `present`. This name was chosen because it lies close to the domain, where this value represents a property that is present. The next three lines define values that are used throughout a generated MiniZinc model. The fourth line defines the actions array. This array represents which action is used at which point in the route. It has the type `ACTION`, which is an enum type that is generated later in the

RSX Type	MinicZinc Type
`int`	`intMinValue..intMaxValue`
`nat`	`0..intMaxValue`
`bool`	`bool`
`unit`	`UNIT`

Table 4.12: The RSX types and their corresponding MiniZinc types.

program. The last two lines of the prelude define usability constraints. The first defines that the last action must always be a `ToSkip` action. The `ToSkip` action is an action that is always added to the RSX model in the explication stage of the compiler. Therefore, its actual definition will be generated later in the program. It is required that this action is always the last action, due to the way constraints over actions are defined. The second constraint defines that whenever a `ToSkip` action is used in the route, the rest of the actions in the route must also be a `ToSkip` action. This reduces the number of equivalent solutions. If this constraint would not be added, a `ToSkip` action could be placed in between any two other actions, which would generate a new solution, which is equivalent to the solution without the `ToSkip` action in between.

Properties

A property in RSX gets translated into two MiniZinc statements. The first defines a variable for the property. This is always an array of optional values with a length equal to the maximum number of actions. The type of the array is dependent on the type of the property in RSX. Table 4.12 shows the RSX types and their corresponding MiniZinc types. For integers and natural numbers, the `intMinValue` and `intMaxValue` parameters are used. These parameters are defined in the prelude of the MiniZinc code that is generated. It was chosen to limit the range of integers and natural numbers because having restricted domains for these types increases performance. The MiniZinc `UNIT` type is also defined in the prelude. The other MiniZinc item that is added for each property is a constraint. This constraint says that the value at the first position of the property array is always the absent value. The reason is that we have to define a starting value for a property, to prevent MiniZinc to choose any value valid for that type. The absent value is convenient because this value is valid for every type of property. An example of the translation of RSX properties to MiniZinc, for each of the different property types, is shown in Figure 4.13.

Actions

Actions are transformed into MiniZinc code by adding the action name to the `ACTION` enum and translating its members. The `ACTION` enum is a type that is added to MiniZinc which holds all possible actions that can be chosen from.

The rest of the action is translated based on its members. An action can have two possible members. A constraint or a preserve expression. Constraint expressions get translated into MiniZinc by translating their expression into a MiniZinc expression. This MiniZinc expression is then used in a MiniZinc constraint. This constraint says that the expression must hold whenever the action that the RSX constraint is defined in, is chosen in the route. An example of this translation is shown in Figure 4.14. It would be redundant to give an overview of how all RSX expressions get translated to MiniZinc because most translations are straightforward. However, it is interesting to see how the `.in` and `.out` expressions get translated. These expressions reason about a specific property. For the `.in` expression, we want to look at the value of the property at the moment of action. Therefore, this gets translated into the value of the property at the i-th position. For the `.out` expression, we want to look at the value of

the property at the moment after the action. Therefore, we translate this to the value of the property at position $i + 1$.

A preserve expression gets translated to a similar constraint in MiniZinc. Here, we say that the value of the property at the i-th position must be the same as the value at position $i+1$. This can also be seen in Figure 4.14, where the property `examplePreserved` gets preserved in the `ToExample` action.

Intent

Intent gets translated similarly as action constraints. The expression that is defining the intent also directly gets translated into a MiniZinc expression. A constraint is added which says that this expression must hold at the last position in the route. An example of the intent translation is shown in Figure 4.15. Note that we have to use an `.in` expression in the intent to ensure that we reason about the property value at the moment of intent.

Devices

The translation of devices into MiniZinc code only consists of the translation of the device paths. These paths define the order of actions that needs to be followed within a specific device. The way to model this in MiniZinc is by adding a constraint that states that whenever an action in a path is chosen, the next action must be the next one in the path. This can be done using a bi-implication because this also holds in the opposite direction. An example of this translation can be seen in Figure 4.16. For this specific example, only a single constraint must be added. The reason is that there is only a single path in the device which has a constraint on the order of actions. Also note that due to the instantiate pass earlier in the compiler, the `ToExample1` action has two instantiations. This ensures that we can still use the `ToExample1` action in the device, without having to use the `ToExample2` action after it. It then chooses the other instantiation of the action, instead of the instantiation which is restricted by the constraint.

Inline

The last RSX top-level definition that needs to be translated into MiniZinc is the inline definition. When two devices are inline, it means that the latest action in a path in the first device must be followed by the first action in one of the paths in the following device. This can be translated to MiniZinc by adding a constraint that says that if one of the last actions of the paths of the first device is chosen in the route, then one of the first actions of the paths of the following device must be chosen. This can be a bi-implication because this also holds the other way around. An example of this translation is shown in Figure 4.17. In this example, one path of the first device has an `exit` on its path. This means that, if this path is chosen in the route, there is no need to go to another action in the other device. This can be achieved because every action in the device is instantiated differently. Therefore, we just add a constraint on the action instantiations which do not have an exit. The instantiations that do have an exit do not get this constraint.

In the example of Figure 4.17, there is also a second constraint added. This constraint says that the first actions which occur on the paths of the second device, cannot occur in the first position of the route. It is important to generate this constraint because these actions can only occur after an action of the first device has occurred. Therefore, there is no possibility that an action of the second device occurs in the first position of the route.

```
device Example1 {
  path
  path ToInstantiate
}

device Example2 {
  path ToInstantiate
  path ToInstantiate -> exit
}

action ToInstantiate {
  constraint true
}

action ToNeverUse {
  constraint false
}
```

(a) The RSX model that gets instantiated.

```
device Example1 {
  path Example1_ToBypass_0
  path Example1_ToInstantiate_0
}

device Example2 {
  path Example2_ToInstantiate_0
  path Example2_ToInstantiate_1 -> exit
}
action Example1_ToBypass_0() {}
action Example1_ToInstantiate_0() {
  constraint true
}
action Example2_ToInstantiate_0() {
  constraint true
}
action Example2_ToInstantiate_1() {
  constraint true
}
```

(b) The instantiated actions of the RSX model.

Figure 4.8: Example of how RSX actions get instantiated during the instantiation pass of the RSX compiler.

```
device Example {
  path ToUniquify1
  path ToUniquify2
}

action ToUniquify1(unique: int) {
  constraint unique > 2
}

action ToUniquify2(unique: int) {
  constraint unique > 2
}
```

(a) The RSX model that gets instantiated.

```
device Example {
  path Example_ToUniquify1_0
  path Example_ToUniquify2_0
}

action Example_ToUniquify1_0(Example_ToUniquify1_0_unique: int) {
  constraint Example_ToUniquify1_0_unique > 2
}

action Example_ToUniquify2_0(Example_ToUniquify2_0_unique: int) {
  constraint Example_ToUniquify2_0_unique > 2
}
```

(b) The instantiated actions of the RSX model.

Figure 4.9: Example of how RSX action parameters get uniquified during the uniquify pass of the RSX compiler.

```
property notPreserved: int
property preserved: int

device Example {
  path ToExplicate
}

action ToExplicate {
  constraint preserved.in < 2
  constraint notPreserved.in < 2
  constraint notPreserved.out == 4
}
```

(a) The RSX model that gets explicated.

```
property notPreserved : int
property preserved : int

device Example {
  path Example_ToExplicate_0
}

action ToSkip() {
  preserves notPreserved
  preserves preserved
}
action Example_ToExplicate_0() {
  constraint preserved.in < 2
  constraint notPreserved.in < 2
  constraint notPreserved.out == 4
  preserves preserved
}
```

(b) The explicated version of the RSX model.

Figure 4.10: Example of how RSX gets explicated during the explicate pass of the RSX compiler.

```
enum UNIT = {present};
int : maxActions = 10;
int : intMinValue = -intMaxValue;
int : intMaxValue = 10000;
array[1..maxActions] of var ACTION : actions;
constraint actions[maxActions] == ToSkip;
constraint forall(i in 1..maxActions - 1)(actions[i] == ToSkip -> actions[i + 1]
↪  == ToSkip);
```

Figure 4.11: The MiniZinc prelude. This prelude is automatically added to every generated MiniZinc program.

```
property exampleInt: int
property exampleNat: nat
property exampleBool: bool
property exampleUnit: unit
```

(a) The RSX properties that get generated in MiniZinc.

```
array[1..maxActions] of var opt intMinValue..intMaxValue : exampleInt;
constraint exampleInt[1] == <>;
array[1..maxActions] of var opt 0..intMaxValue : exampleNat;
constraint exampleNat[1] == <>;
array[1..maxActions] of var opt bool : exampleBool;
constraint exampleBool[1] == <>;
array[1..maxActions] of var opt UNIT : exampleUnit;
constraint exampleUnit[1] == <>;
```

(b) The generated MiniZinc code for the properties defined in RSX.

Figure 4.13: Example of how RSX properties get translated into MiniZinc code.

```
property exampleConstraint: int
property examplePreserved: int

action ToExample {
  constraint exampleConstraint.in < 3
  constraint exampleConstraint.out == 3
  preserves examplePreserved
}
```

(a) The RSX action that gets generated in MiniZinc.

```
enum ACTION = {ToSkip};

constraint forall(i in 1..maxActions - 1)(actions[i] == ToExample ->
↪  exampleConstraint[i] < 3);
constraint forall(i in 1..maxActions - 1)(actions[i] == ToExample ->
↪  exampleConstraint[i + 1] == 3);
constraint forall(i in 1..maxActions - 1)(actions[i] == ToExample ->
↪  examplePreserved[i] == examplePreserved[i + 1]);
```

(b) The generated MiniZinc code for the action defined in RSX.

Figure 4.14: Example of how RSX actions get translated into MiniZinc code. Note that this example leaves out devices, action instantiation and property translation, in order to give a better overview of the transformations specific for actions.

```
intent exampleIntent.in > 2
```

(a) The RSX intent that gets generated in MiniZinc.

```
constraint forall(i in maxActions..maxActions)(exampleIntent[i] > 2);
```

(b) The generated MiniZinc code for the intent defined in RSX.

Figure 4.15: Example of how RSX intent gets translated into MiniZinc code. Note that this example leaves property definitions, in order to give a better overview of the transformations specific for intent.

```
device Example {
  path
  path ToExample1
  path ToExample1 -> ToExample2
}

action ToExample1 {}

action ToExample2 {}

action ToExample3 {}
```

(a) The RSX device that gets generated in MiniZinc.

```
device Example {
  path Example_ToBypass_0
  path Example_ToExample1_0
  path Example_ToExample1_1 -> Example_ToExample2_0
}
action Example_ToBypass_0( ) {}
action Example_ToExample1_0( ) {}
action Example_ToExample1_1( ) {}
action Example_ToExample2_0( ) {}
```

(b) The instantiated RSX device that gets generated in MiniZinc.

```
constraint forall(i in 1..maxActions - 1)(actions[i] == Example_ToExample1_1 <->
↪  actions[i + 1] == Example_ToExample2_0);
```

(c) The generated MiniZinc code for the device defined in RSX.

Figure 4.16: Example of how an RSX device gets translated into MiniZinc code.

```
inline Example1 -> Example2
device Example1 {
  path ToExample1 -> exit
  path ToExample1
  path ToExample2
}

device Example2 {
  path ToExample1
  path ToExample2
}

action ToExample1 {}

action ToExample2 {}
```

(a) The RSX inline devices that gets generated in MiniZinc.

```
inline Example1 -> Example2
device Example1 {
  path Example1_ToExample1_0 -> exit
  path Example1_ToExample1_1
  path Example1_ToExample2_0
}
device Example2 {
  path Example2_ToExample1_0
  path Example2_ToExample2_0
}
action Example1_ToExample1_0() {}
action Example1_ToExample1_1() {}
action Example2_ToExample2_0() {}
```

(b) The instantiated RSX inline devices that gets generated in MiniZinc.

```
constraint forall(i in 1..maxActions - 1)(actions[i] == Example1_ToExample1_1 \/
↪  actions[i] == Example1_ToExample2_0 <-> actions[i + 1] ==
↪  Example2_ToExample1_0 \/ actions[i + 1] == Example2_ToExample2_0);
constraint not (actions[1] == Example2_ToExample1_0 \/ actions[1] ==
↪  Example2_ToExample2_0);
```

(c) The generated MiniZinc code for the inline devices defined in RSX.

Figure 4.17: Example of how RSX inline devices get translated into MiniZinc code.

Chapter 5

Evaluating RSX

The development of a domain-specific or a general language is not just a question of thinking about a solution, building it, and then using it. It involves a whole process of evaluation as well. Wirth [27] articulates this well in his paper "On the Design of Programming Languages". There he says: "...when the project is at its end, carefully reassess it, recognise that many aspects could be improved, and do it all over again." This is of course also true for RSX. We must evaluate the language and see where it can be improved.

5.1 Evaluation Aspects

The language can be evaluated on an almost infinite amount of aspects. Given the limited resources, we decided to evaluate the language on the three aspects we thought to be the most interesting given the current state of research. The language is developed completely from the ground up. Therefore, it is a logical consequence that there is uncertainty about the choices made in the development of the language. Given that the language is mainly used in an industrial environment, we needed to make sure that the language can also be used in this environment and that it indeed solves the routing problem we described in Chapter 2. We evaluated the following aspects of the language.

Language Coverage The language needs to facilitate the modeling of realistic cases. In this evaluation, we evaluate whether it is possible to model these cases and which language aspects can be enhanced to improve the modeling experience. This evaluation is essentially an answer to whether RSX solves the problem described in Chapter 2. Observations made during this evaluation are denoted as *COVERAGE i*, where *i* is the number of the observation.

Language Accuracy The language needs to be complete and correct. This means that the language does not say there is no route available when there is, and that the language does not say there is a route available when there is not. In this evaluation, we evaluate whether this is the case for RSX. Observations made during this evaluation are denoted as *ACCURACY j*, where *j* is the number of the observation.

Language Performance The language is designed as a proof-of-concept and does not have specific performance criteria. However, we want to see how this proof of concept performs and if it can be used in an industrial context. This means it should not take a whole day to come up with a solution. If the total time is in the order of a few seconds, we consider it to be usable. In this evaluation, we will evaluate the runtime performance of RSX by running benchmarks on different cases. Observations made during this evaluation are denoted as *PERFORMANCE k*, where *k* is the number of the observation.

During the evaluations of these aspects, some general observations were made, which are not part of one of these categories. These are denoted as *GO l*, where *l* is the number of the observation.

5.2 Language Coverage

The goal of a domain-specific language is to model the most prevalent concepts of the domain as closely to the domain as possible. To ensure that the DSL indeed covers these concepts correctly, we evaluated the coverage of the language. With the term *coverage* we mean the extent in which the concepts that exist in the domain, can be modeled in the language. When a concept is well covered in the language, it is straightforward to express it in the language. If a concept is not well covered, it is not possible to express it in the language, or it takes a lot of boilerplate code to express the concept.

5.2.1 Setup

To test the coverage of the language, we used a method called *think-aloud co-design sessions* [11]. This method has been used at Canon Production Printing before and has shown to be successful in evaluating the coverage of a DSL [10]. The first participant in these sessions is the developer of the language. The second participant in these sessions is a domain expert. This domain expert has been involved in the development of the language as well. In the last session, an extra participant joins the session, which is a product expert. The product expert is a developer of the current control software which was used in the printing systems.

The goal of the sessions is to gather data on the domain coverage of the language. Therefore, during the sessions, the language is used to implement realistic cases. After implementation, the process of implementation and the implementation itself were evaluated by the participants. If required, the developer of the language makes changes to the language intending to improve the implementation of the points discussed. After that, this new iteration is evaluated in the same way as the previous one and the effectiveness of the improvements is discussed as well.

Performing the think-aloud co-design sessions can be described using the following protocol:

1. The domain expert selects one or multiple cases to model in the RSX language. This expert tries to select a case in increasing order of complexity.
2. The participants perform an iteration in which the developer tries to model the selected case, guided by questioning the domain expert about the characteristics of the case.
3. If changes were made to the language in the previous iteration, the participants evaluate these changes. This is done by using the features in the language that were changed.
4. The domain expert, the product expert, and the developer write down the decisions they made during the modeling of the case.
5. The participants evaluate the correctness of the model by running the generated code in the MiniZinc editor and making any changes to the RSX model if needed.
6. The participants discuss the decisions and observations made and decide whether there is a need to change the language to make it easier to implement the case.
    - If the language needs changes, the developer makes the discussed changes to the language.
    - If the language does not need changes, the participants continue with the process.

7. The participants repeat the process until the domain expert concludes that a production setup is modeled with a level of detail that is sufficient for a prototype of control software.

5.2.2 Results

In total, six sessions were performed by the participants. In these sessions, a total of five different cases were modeled. During the evaluation, the language significantly changed based on the observations made during the sessions. Therefore, some of the concepts which we presented in Chapter 4 did not exist during the earlier evaluation sessions. We considered it not useful to present all the individual language versions in this chapter, and we therefore only define the significant changes to the language.

Session 1

Cases In the first session, the participants started by modeling two different cases. In the first case, a simple setup was modelled in which there were only three actions: `ToPick`, `ToOrientate` and `ToTrim`. The goal of this first case was to model a route in which the intent could only be achieved by a combination of the available actions, but with more than one correct solution. The concept of devices and paths did not exist yet during this evaluation.

In the second case, the participants modeled a set of actions that could produce a simple product of a stack of printed sheets that were bound by stitches. Such a product is called `loose-leaf`. This case was chosen because it is one of the simplest products that can be produced by a digital printing system.

Observations Each property in RSX gets translated to an array in MiniZinc. This array holds the value of the property at each step in the route. The problem with this translation is that for some steps of the route, properties might not have a value. The most obvious example of this is the value of a property in the first step of the route. No properties have a value in this first step, because they do not get assigned an initial value in RSX. To model this absence of a value, each property is modeled as an array of optional values and the initial value of a property is set to absent. An example of a part of the translation of a property is shown in Figure 5.1.

```
property example: int
```

(a) Modelling of a property in RSX. The property has the name `example` and is of type `int`.

```
array[1..maxActions] of var opt intMinValue..intMaxValue : example;
constraint example[1] == <>;
```

(b) Part of the generated MiniZinc code for an example property in RSX. The property gets translated to an array of optional integers, where the initial value is absent.

Figure 5.1: Translation of a single property in RSX to MiniZinc in the first iteration of RSX.

In the domain, properties get a value when an action introduces them. In RSX, this is modeled by adding a constraint in an action on its output value. We call this the *introduction* of a property. Figure 5.2 shows an example of such an introduction in RSX and the translation of this constraint to MiniZinc.

```
1 property example: int
2
3 action ToIntroduce() {
4   constraint example.out == 2
5 }
```

(a) Modelling of the introduction of a property in RSX. In the `ToIntroduce` action, we specify a value for the `example` property.

```
1 array[1..maxActions] of var opt intMinValue..intMaxValue : example;
2 constraint example[1] == <>;
3 constraint forall(i in 1..maxActions - 1)(actions[i] == ToIntroduce -> example[i
  ↪  + 1] == 2);
```

(b) Translation of the RSX code in MiniZinc. The constraint added to the `ToIntroduce` action in RSX, gets translated to a MiniZinc constraint in line 3.

Figure 5.2: Translation of the introduction of a property in RSX to MiniZinc in the first iteration of RSX.

As can be seen in this figure, the translation of the constraint in RSX does not put a restriction on the value of property in the i-th position and only a restriction on the value in the position $i + 1$. This means that there is no constraint here that prevents the property from already having a value in the i-th position. This is not correct, because when an action is *introducing* a property, the property does not have a value before the action. In this iteration of RSX it is not possible to model this (*COVERAGE 1*).

The introduction of properties is not the only aspect of properties that is impossible to model in RSX. In the domain, some properties might have a value, while after the execution of an action, this property does not exist anymore. For example, when we trim off a part of a stack, the stitches in that stack could be removed. This is what we call the *removal* of a property. In the constraint model, this could be modeled by setting the value of the property to be absent in the step after the action. However, in this iteration of RSX it is not possible to get such a constraint generated in MiniZinc (*COVERAGE 2*).

Two other observations that the participants made in this iteration were more general. The first is that RSX generates a MiniZinc model with an unnecessary amount of possible solutions. The reason is that there is no restriction on the position of the `ToSkip` action in the generated solution. This means that this action can be put in between any two other actions, which would generate a new solution. However, this new solution is equivalent in the domain to the solution without the `ToSkip` action (*GO 1*).

The other general observation that the participants made is that the way of defining the intent does not have a natural syntax. An example of the way that intent is described is shown in Figure 5.3. As can be seen there, to specify the intent of the property `example`, the `in` word needs to be placed behind it. The reason is that this makes the translation of RSX to MiniZinc easier. However, this is not idiomatic to the concept of intent in the domain (*GO 2*).

Language Evolution Based on the observations of the first session, we made two changes to the language before executing a new iteration. The first change was to add an extra constraint to the place where `ToSkip` actions can occur. This constraint enforces all `ToSkip` actions to be

```
1 property example: int
2
3 intent example.in < 3
```

Figure 5.3: Specification of the intent on a property in the first iteration of RSX.

the last actions of the route. This way, it is no longer possible to place a `ToSkip` action in between two other actions and therefore create a new equivalent solution. The generated constraint can be seen on line 1 in Figure 5.4b.

The other change that was implemented, tries to tackle the problem of introducing properties (*COVERAGE 1*) and removing properties (*COVERAGE 2*). To make it possible to let an action introduce or remove a property in RSX, two new expressions were introduced to the language. These new expressions are shown in Figure 5.4. The `introduce` keyword is used to introduce a property in an action. This expression requires a value to be given to the property after introduction. The `remove` keyword is used to remove a property in an action. The constraints that are generated from these new concepts are shown in Figure 5.4b on lines 5-6 and lines 7-8 respectively. This translation happens directly from the RSX model to MiniZinc. One might expect that we would use a desugaring step for this. However, in this version of RSX, there was no way to express the presence or absence of a property yet, so the only way to translate the introduction and removal concepts was by a direct translation from RSX to MiniZinc.

Session 2

Cases In the second session, the participants tried to remodel the second case of the first session. In this case, another setup to make loose-leaf products was modeled, this time using the newly introduced concepts of introducing and removing properties.

Observations The newly introduced concepts allow the actions to correctly model the introduction and removal of a property. However, now that this can be done correctly, it exposes a problem with properties that the participants had not observed. This problem occurs when trying to model properties that do not have a specific value. In the domain, some properties only exist or do not exist on a product and do not hold a value. This cannot be modeled in this iteration of RSX. One could think that this is achievable using boolean properties. However, this is not the case. Note that boolean properties in RSX get translated to an optional boolean array in MiniZinc. This means that every value of the array can have three possible values: `<>`, `true` and `false`. That is more than the required two possible values, namely either present or absent (*COVERAGE 3*).

Another observation made, is that the expressions used to reason about the introduction and removal of a property in an action, are too restrictive. In some actions, it is possible that the property is only introduced when not already introduced in a previous action or only removed whenever the property is present and not already removed in a previous action. This cannot be expressed in the language using these newly introduced expressions (*COVERAGE 4*).

Language Evolution After this session, it was decided to change two major aspects of the language. In the first change, we tried to ensure that the concepts of introduction and removal of properties are more flexible (*COVERAGE 4*). To achieve this, we generalized the concepts of introduction and removal to expressions that describe whether a property is restricted to

```
1 property example: int
2
3 action ToIntroduce() {
4   introduce example = 2
5 }
6
7 action ToRemove() {
8   remove example
9 }
```

(a) Modelling of the introduction of a property in RSX. In the `ToIntroduce` action, we specify a value for the `example` property.

```
1 constraint forall(i in 1..maxActions - 1)(actions[i] == ToSkip -> actions[i + 1]
  ↪  == ToSkip);
2
3 array[1..maxActions] of var opt -10000..10000 : example;
4 constraint example[1] == <>;
5 constraint forall(i in 1..maxActions - 1)(actions[i] == ToIntroduce ->
  ↪  absent(example[i]));
6 constraint forall(i in 1..maxActions - 1)(actions[i] == ToIntroduce -> example[i
  ↪  + 1] == 2);
7 constraint forall(i in 1..maxActions - 1)(actions[i] == ToRemove ->
  ↪  occurs(example[i]));
8 constraint forall(i in 1..maxActions - 1)(actions[i] == ToRemove ->
  ↪  absent(example[i]));
```

(b) Translation of the RSX code in MiniZinc. The constraint added to the `ToIntroduce` action in RSX, gets translated to a MiniZinc constraint in line 3.

Figure 5.4: Use and translation of the `introduction` and `remove` expressions in the second iteration of RSX.

be absent or present. An example of these new expressions can be seen in Figure 5.5a on lines 4 to 5 and 9 to 10. These expressions are directly translated into MiniZinc code by using the built in `occurs` and `absent` expressions. An example can be seen in Figure 5.5b lines 4 to 7.

The second change tries to solve the problem in which it is not possible to model properties without value (*COVERAGE 3*). The solution to this problem is the introduction of a new type to the language. This type is called the *unit* type. In other programming languages, such a type is used to describe a type that can only have one possible value. The same holds for the unit type in RSX. This type is used to describe a property that does not hold a value and can only exist or be absent. The only way that can be reasoned about such a property is by the newly introduced expressions for the absence and presence of a property. An example of this new property type is shown in Figure 5.5. The type is annotated with the `unit` keyword and gets translated to an optional MiniZinc enum with just a single value.

```
property example: unit

action ToIntroduce() {
  constraint example.in.absent
  constraint example.out.present
}

action ToRemove() {
  constraint example.in.present
  constraint example.out.absent
}
```

(a) Modelling of the introduction of a property in RSX which has a unit type. In the `ToIntroduce` action, we do not specify a value for the `example` property, but merely restrict that the value should be present.

```
enum UNIT = {present};
array[1..maxActions] of var opt UNIT : example;

constraint forall(i in 1..maxActions - 1)(actions[i] == ToIntroduce ->
↪ absent(example[i]));
constraint forall(i in 1..maxActions - 1)(actions[i] == ToIntroduce ->
↪ occurs(example[i + 1]));
constraint forall(i in 1..maxActions - 1)(actions[i] == ToRemove ->
↪ occurs(example[i]));
constraint forall(i in 1..maxActions - 1)(actions[i] == ToRemove ->
↪ absent(example[i + 1]));
```

(b) Translation of the RSX unit property in MiniZinc. The constraints added to the `ToIntroduce` action in RSX, get translated directly to a MiniZinc constraint in line 3.

Figure 5.5: Use and translation of the `present` and `absent` expressions in the third iteration of RSX.

Session 3

Cases In the third session the participants implemented the same case as in the second session, this time with the newly introduced changes of the unit types and the absence and presence expressions. The goal was to evaluate whether these changes could indeed solve the problems that were encountered in earlier evaluation sessions.

Observations The newly introduced unit types work well for modeling properties that do not have value and using the newly introduced presence and absence expression allows for idiomatically constraining the presence and absence of those properties (*COVERAGE 5*).

The absence and presence expressions on properties can be used to define the introduction and removal of properties. It also provides more flexibility than the earlier used introduction and removal expressions (*COVERAGE 6*).

However, the participants observed that it would be nice if it was possible to automatically have constraints added to these properties, whenever it is logical to do so. For example,

if a constraint is added to the value of a property, it follows that the property should also be present. This now requires the manual work of adding the presence constraint to the model (*COVERAGE 7*).

The participants make the observation that some properties could be grouped. They are defined as different properties but say something about the same part of the product. For example, the property for stack width and the property for stack height both say something about the dimensions of a stack. Furthermore, in the domain, these cannot exist without each other. In the language they are seen as two different, independent properties (*COVERAGE 8*).

Because the implemented case can result in many solutions, which are all valid, the participants observe that it is useful to add a minimization target on the number of actions used in the route. This way, the solver in MiniZinc only returns the route with the fewest number of actions. Even though this is a simple optimization, the participants observe that this is often a good solution (*COVERAGE 9*).

Language Evolution After this session, no changes were made to the language.

Session 4

Cases In this session, a case was modeled for which some actions were *inline*. This means that the order of actions is partially fixed. It is common in the domain that some actions always occur before or after another. For example, there could be a device that makes booklets and after a folding action, a stitching action always occurs. In the earlier cases, such restrictions were not modeled and every action could occur before or after any other action.

Observations To model the fact that some actions should come before others, we used properties in RSX which modeled the location of an action. A very basic example that shows how this was done, can be seen in Figure 5.6. In this figure, the `ToDo1` action always has to be chosen before a `ToDo2` action can be chosen. The reason is, that the `position` property is only set in the first action and required to be set in the second action. If the first action would not occur, the second action cannot occur either. The observation was made that this is a viable way to model a fixed order of actions (*COVERAGE 10*).

Language Evolution After this session, no changes were made to the language.

```
1  property position: int
2
3  action ToDo1 {
4    constraint position.in.absent
5    constraint position.out == 1
6  }
7
8  action ToDo2 {
9    constraint position.in == 1
10   constraint position.out.absent
11 }
```

Figure 5.6: Specification of an RSX property which is used to determine a fixed order in actions.

Session 5

Cases In this session, the participants implemented a part of a more realistic setup in comparison to earlier cases. For this, the domain expert made an overview of a printer with the possible finishing devices which could be added to it. For each of these devices, the possible paths of actions were selected. The devices that were then chosen to be modeled are shown in Figure 5.7. The devices that were chosen are all *inline*. This means that the order of devices is fixed. For each of the devices, there is also a set of paths of actions that are possible in the device. The special actions `ToBypass` and `ToDeliver` do not change the product in any way but are used respectively to model the fact that a device is bypassed and that the product is delivered. By delivery of a product, we mean that the product leaves the inline setup and that the product can be physically taken from the device.

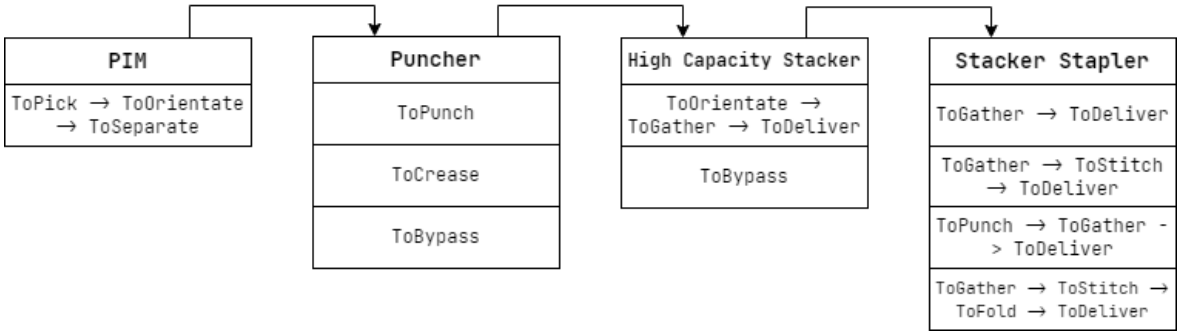

Figure 5.7: The devices that were modelled in the fifth language coverage evaluation session.

Observations The first observation that was made by the participants, was that in this case, the actions have less flexibility than in earlier cases. Not only is the order of actions fixed, but the order of devices and the possible paths is also something that needs to be considered. Some actions can only be used in a specific device and some actions can be used in multiple devices. The RSX language leaves it up to the writer of the model how to model these constructs (*COVERAGE 11*).

The participants tried to implement the restrictions on the actions in the same way as in the previous session. Using a property to define the position of the product in a device, the actions could be restricted to only occur in a specific position. Figure 5.8 shows an example of how this was done for the `ToOrientate` actions. The constraint that is defined on lines 8 to 18, says that the `ToOrientate` action can occur in two places. As can be seen in Figure 5.7, the action can occur in the `PIM` or the `High Capacity Stacker`. This is represented by the property `INLINE_POSITION`, which has to have the value 1 or lower than 3. When this property has the value 1, it has to appear in a specific position in the `PIM`, which is denoted by the `PIM_POSITION` property. When the `INLINE_POSITION` has a value lower than three, then the action must be at a specific position in the `High Capacity Stacker`. This is modeled using another property `HCS_POSITION`. Note that each of these properties must define all possible values.

The participants note that it is very tedious to define such constraints for each action. It is difficult to reason about the position of an action in this way, and it is easy to make mistakes. Modelling an inline setup this way is therefore not desirable (*COVERAGE 12*).

Language Evolution The changes that were made after this session focus on making inline setups easier to model in RSX. As observed in this session, there is nothing in the language to explicitly define such restrictions and it is up to the programmer to find a viable solution (*COVERAGE 11*). Using this viable solution was tedious for the participants (*COVERAGE 12*). The participants, therefore, wanted to implement a new concept in the language which allowed for easier modeling of these restrictions on the order of devices and actions.

```
property INLINE_POSITION: int
property PIM_POSITION: int
property HCS_POSITION: int

action ToOrientate {
    constraint INLINE_POSITION.in == 1
           and INLINE_POSITION.out == 1
           and PIM_POSITION.in == 1
           and PIM_POSITION.out == 2
           and HCS_POSITION.out.absent
            or INLINE_POSITION.in.present
           and INLINE_POSITION.in < 3
           and INLINE_POSITION.out == 3
           and HCS_POSITION.in.absent
           and HCS_POSITION.out == 1
           and PIM_POSITION.out.absent
}
```

Figure 5.8: Example of how RSX properties can be used to determine a fixed order in actions and device. Note that the properties which were not used for restricting the position of the action, are left out for this example.

A starting point for these changes was the fact that the language provided no way to define which actions could occur in which device. None of the actions were part of a specific device. Furthermore, restricting the device in which an action could be used, using concepts that already existed, a very tedious process. Therefore, in the next iteration of the language, the concept of a *device* was introduced. An example of how this new concept can be used in the language is shown in Figure 5.9. For each device, the programmer defines a name and a set of paths. These paths consist of actions defined in the RSX model. A path can have zero entries, which represents a bypass of this device.

Next to a fixed order of actions within a device, there can be also a fixed order in devices. Therefore, the *inline* concept was introduced. This concept models the fact that some devices are set up to be inline. Therefore, the actions in these devices are restricted to also follow this specific order. An example of this new concept can be seen in Figure 5.10. We define two devices: `Example1` and `Example2`. These devices are inline. Therefore, the device `Example2` can only occur after device `Example1`. We describe this as an inline path on line 3. Note that we also used another expression in a device path of device `Example1`. The `exit` keyword is used to describe a path that can leave the inline setup. This means that after the `ToIntroduce` action, the device `Example2` does not have to occur in the route. If we left out this path, the `ToRemove` action would always have to occur after the `ToIntroduce` action. An important aspect that can be derived from this example, is that the inline path and the paths in devices are not optional. If a device or action occurs in a path, a following device or path must also occur. The only exception is when a path contains the `exit` keyword.

Session 6

Cases In this last session, the participants modeled the same case as in the previous session. However, now the newly introduced concepts of devices and inline devices were used. The participants wanted to evaluate whether these new concepts were indeed a good addition to the language.

```
property example: int

device Example {
  path ToIntroduce -> ToRemove
  path ToIntroduce
  path
}

action ToIntroduce {
  constraint example.in.absent
  constraint example.out == 2
}

action ToRemove {
  constraint example.in.present
  constraint example.out.absent
}
```

Figure 5.9: Example of the device concept introduced in the sixth iteration of the RSX language.

```
property example: int

inline Example1 -> Example2

device Example1 {
  path ToIntroduce
  path ToIntroduce -> exit
}

device Example2 {
  path ToRemove
}
```

Figure 5.10: Example of the inline concept introduced in the sixth iteration of the RSX language. Note that the action definitions are left out of this example.

Observations The first observation the participants made describes a concern raised by the product expert participant. In this version of the language, only actions can add preconditions on the product that it performs its changes on. However, it sometimes depends on the device the action is used in, and what the exact precondition is. For example, an action can only handle a stack of a certain size. The actual maximum stack size can be dependent on the device the action is used in. Although it is not possible to define this in the language at the moment, this can be solved by defining a different action for each device. However, this negates the advantages of having to define actions with common behavior only once (*COVERAGE 13*).

Another observation the participants made, was that defining the paths in devices and

paths of inline devices is intuitive. They make it easy to describe the capabilities of a device and the different options in the device (*COVERAGE 14*).

The developer of the language and the domain expert observe that modeling the case is easier compared to modeling this case in MiniZinc. A lot of heavy lifting is done by the language. For example, actions only need to be defined once and are then automatically instantiated for the path of a device. For devices that have many paths and actions that are used in many devices, this reduces the time of modeling significantly (*COVERAGE 15*).

The last observation made by the participants is that the concept of the unit type is not immediately clear to some participants. The question that arises is what the difference is with a boolean value (*GO 3*).

Language Evolution No language changes were made after this session.

5.2.3 Conclusions

We evaluated the coverage of domain features in the language coverage evaluation. During six think-aloud co-design sessions, the participants observed that the language could indeed be used to model eight cases. These cases are representative for the context of our industrial partner. We chose cases with increasing complexity to be able to iteratively improve the language after each evaluation session.

The participants observed that the language features for devices, actions, and inline devices, help in modeling the problem (*COVERAGE 14*). The modeling of properties in RSX is done using only primitive types and does not allow for modeling composite types. This does not mean composite properties cannot be modeled, but it makes modeling of these properties more involved (*COVERAGE 8*).

We evaluated that the use of RSX is an advantage over modeling the problem directly in a constraint language. Modeling the problem in MiniZinc directly is much more involved than in RSX, because of the higher level of abstraction in RSX (*COVERAGE 15*). This is important, because of the large diversity in devices, users of devices, and existing route constraints.

RSX can provide one or multiple routes to the cases we evaluated. When there is only a single route for a model, the solution space is small, and it is immediately clear what route to choose. If there are multiple routes, the solution space is larger, and it is not directly apparent which route to choose. To aid in this process, the MiniZinc model generated by RSX can be extended to optimize the solution based on a cost function. The participants observed that this makes it easier to choose from one of the provided solutions (*COVERAGE 9*).

5.3 Language Accuracy

The language as described in this thesis is implemented fully in the Spoofax language workbench [17]. This implementation needs to be correct and complete. Correct means that what we specify about the language and the model behind it, must follow from the implementation. Complete means that the implementation must do everything the specification says it does. In general, proving the correctness of a software program is a a profession in and of itself. Also, for our language implementation, providing a proven correct implementation is not within the scope of this thesis. However, we still want to evaluate the correctness and completeness of the implementation.

5.3.1 Setup

To evaluate language accuracy, we implemented a test suite for the implementation. In this test suite, we systematically tested all language features in several categories. These cate-

Category	Number of tests
Parsing	122
Static Semantics	117
Transformation	46
Total	285

Table 5.11: Number of tests per category used to evaluate the RSX language accuracy.

gories test the different parts of the language implementation, but also the implementation of the resulting MiniZinc code. The categories that we tested are:

Parsing Whether programs in the language that follow the RSX grammar parse correctly into an AST and whether programs that do not follow the grammar give an error.

Static Semantics Whether programs that follow the static semantic rules defined for the language pass the static semantics checking and whether programs which do not follow the static semantic rules do not pass the static semantic checking.

Transformation Whether programs that follow the grammar and static semantic rules of the language, get transformed into MiniZinc code correctly.

5.3.2 Results

In total, 285 tests were written, and all passed. The number of tests written per category are shown in Table 5.11 (*ACCURACY 1*). Some tests were written during the language coverage sessions because of bugs in the language. For each bug, a test was added to the test suite (*ACCURACY 2*).

5.3.3 Conclusions

We evaluated the language compiler accuracy by using a test suite. The goal was to have a compiler that is accurate enough to be usable as a prototype. We defined this as that the language compiler should accurately compile normal use cases. We did not specifically test edge cases. The tests cover all the language features at least once on every part of the language compiler, i.e. parsing, static semantics, and transformations of the language.

With a total of 285 tests, we are confident that the language is both correct and complete up to a level that is usable as a prototype (*ACCURACY 1*). Bugs were encountered during the coverage evaluation sessions, which were then covered by writing extra tests (*ACCURACY 2*). The existence of incorrectness or incompleteness of the language is not something that we can eliminate by a test suite alone. Moreover, providing full proof of correctness and completeness is outside the scope of this thesis. For a prototype of a language, the existence of bugs in the compiler is annoying, but unsurmountable. The focus therefore is not to make a fully accurate language compiler.

5.4 Language Performance

The last evaluation that we performed was about the performance of the language. The performance of RSX is determined by two different parts. The first part is the time it takes to compile the language to MiniZinc. The second part is the time it takes to find a solution for the problem described in the generated MiniZinc code.

For both the compilation of the language and the solving of the generated MiniZinc model, we evaluated its runtime performance. We evaluate if we can compile and solve

RSX models in the order of a few seconds. In the industrial context of Canon Production Printing, this is considered a usable time.

5.4.1 Setup

The language performance is evaluated by running benchmarks on four different RSX models. These models are developed during the language coverage session and differ in size and use of language features. The models used and their characteristics are shown in Table 5.12.

Benchmark	Session	Nr. Property	Nr. Action	Nr. Device	Nr. Inline
C1	4	4	6	6	0
C2	3	2	4	4	0
C3	5	13	9	9	0
C4	6	9	10	6	1
S1	5	13	9	9	0
S2	6	9	10	6	1

Table 5.12: The different characteristics of the chosen benchmark cases.

For both the compilation and solving benchmarks, we chose models we already modeled in the language coverage evaluation sessions described in Section 5.2. For the compilation tests, we selected the models that were valid following the latest RSX specification. The reason we did not use all the cases we implemented in the coverage evaluation, is that some concepts were removed or changed in such a way that some cases would not compile anymore. Therefore, it is not possible to get comparable benchmark results using a single RSX specification.

The solving benchmarks were performed on the largest two cases we implemented in the language coverage evaluation session. These are the two most realistic cases at our disposal, regarding the size of these cases. The other cases we modeled, are smaller and model only a part of the finishers in a realistic production setup.

All benchmarks were run using the JMH framework [1]. This framework is specifically built for performance testing of Java applications. All Stratego code compiles to Java, so we used this framework in combination with the compiled Stratego code. Each step of the transformation from RSX to MiniZinc was individually timed using this framework. The benchmarks were run 10 times each, after 10 warm-up iterations. The device used for the benchmarks was isolated from a network and it only ran the operating system, alongside the benchmarks. The device has an Intel Core i7-7700 CPU processor with a clock speed of 3.60GHz. It also has 16GiB of DDR4 RAM.

5.4.2 Results

The exact runtimes for each part of the compilation process are shown in Appendix A, in Table A.1. As can be seen there, the compilation part that generates a MiniZinc string from the MiniZinc AST has the longest runtime (*PERFORMANCE 1*).

The average runtimes for the other compiler stages for each case are shown in Figure 5.13. For clarity, we left out the times for generating a MiniZinc string from the AST. In all four cases, the parsing times take the longest, followed by the translation of RSX to MiniZinc (*PERFORMANCE 2*).

The average runtimes for solving the two different cases are shown in Figure 5.14 and in Table A.2. The solving times for compilation take significantly more time than the actual solving (*PERFORMANCE 3*).

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

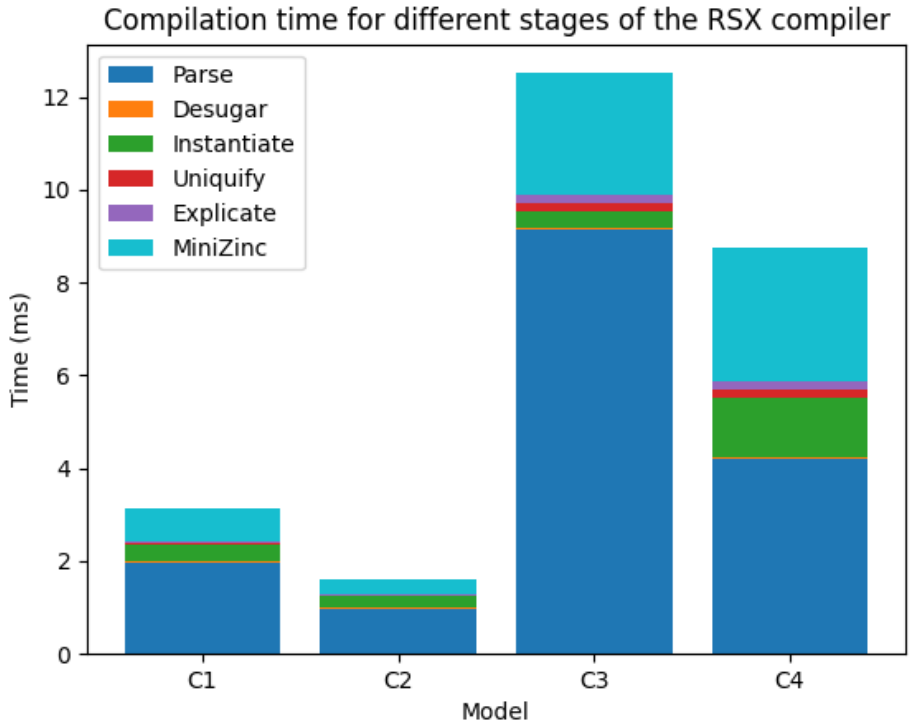

Figure 5.13: The compilation times for four different RSX models, split into the different compiler stages.

The average total time of solving and compilation for both cases is in the order of a few seconds (*PERFORMANCE 4*).

5.4.3 Conclusions

We evaluated the runtime of compilation and solving during the language performance evaluation. The goal was to evaluate whether the language could do so in the order of seconds. In this evaluation, we ran benchmarks on the compilation and solving times of different cases modeled in RSX. These benchmarks show that compiling and solving these cases takes no more than a few seconds (*PERFORMANCE 4*). In these cases, compiling takes more time than solving, and especially generating the MiniZinc string from the MiniZinc abstract syntax tree takes significantly more time (*PERFORMANCE 1*).

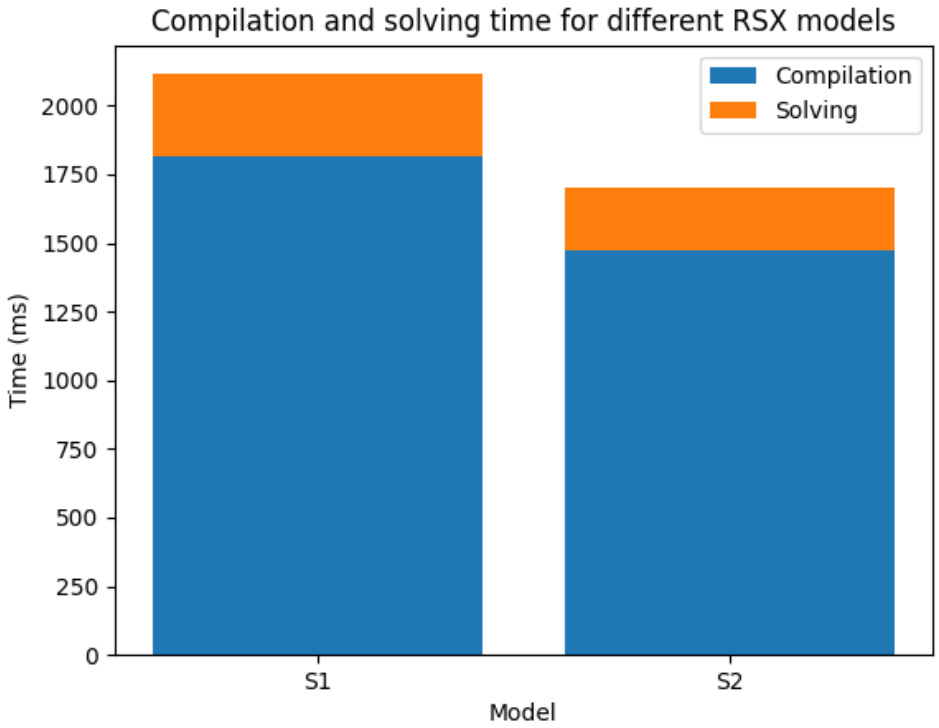

Figure 5.14: The compilation and solving times for two different RSX models, split into the compilation part and the solving part.

Chapter 6

Discussion

6.1 Language Limitations and Design

Complex Typing The implementation of RSX currently uses only primitive types. User-defined types or composite types, i.e. types which combine other types, are not supported. There are cases where it might be useful to have more complex types in the language. For example, in our domain, a sheet is an often-used type. A sheet can have a width, height, and color. Currently, in RSX we have to define these properties of a sheet as three separate properties. There is nothing that binds these properties together. When we want to express that a sheet is present, we have to add a constraint on each of these three properties separately. It is easy to forget such a property.

A solution for this would be to construct a new type called `Sheet`, which groups these properties. An example can be seen in Figure 6.1. There we added a type definition for `Sheet` with the three properties of a sheet. In the action, we now only have to define the presence constraint once for the complete type.

Enumerable Types Apart from complex types, in Figure 6.1 we also show a hypothetical concept of enumerable types. Currently, these types need to be defined by an integer, and we have to define constraints to ensure that this integer stays within a certain range. This is to prevent any invalid values for that type. By defining an enumerable type as an integer, we also do not have a clear way of expressing the underlying value of that integer.

We could extend the language by adding an `enum` type which takes care of these issues. We directly define the underlying values for such a type and the type ensures that we only use one of these values. An example can be seen in Figure 6.1.

Route Limitations The current constraint model assumes that a route is always linear, i.e. always a sequence of actions. However, this does not represent all possible routes in the domain. Some routes split a product halfway through the route, where one part of the product continues on a different route than the other part. It is also possible that a route combines two products into one. An example of this is a perfect-bound book, where the cover sheet takes a different route before it is combined with the book block.

To model such routes, an array representation in MiniZinc is insufficient. A more complex data structure, like a graph, has to be used to represent a route. One concern that arises is that such a data structure increases the size of the solution space. This could negatively influence the solving performance, and it should be researched whether this influence is surmountable.

Device Constraints Devices in RSX define a set of constraints on the possible action paths that can be used within that device. Unlike actions, devices cannot constrain the values of

```
property sheetWidth: int
property sheetHeight: int
property sheetColor: int

action ToTest {
  constraint sheetWidth.in.present
  constraint sheetHeight.in.present
  constraint sheetColor.in.present

  constraint sheetColor.in > 0 and sheetColor.in < 5
}
```

(a) Modelling of complex types in current version of RSX.

```
enum Color { Cyan Magenta Yellow Black }
type Sheet {
  width: int
  height: int
  color: Color
}

property sheet: Sheet

action ToTest {
  constraint sheet.in.present
}
```

(b) Hypothetical modelling of complex types in a future version of RSX.

Figure 6.1: Comparison between modelling of complex types in current version of RSX and hypothetical future version of RSX.

properties in that device. However, in the domain, an action that is used in one device could have different constraints when used in another device. For example, some devices can gather 100 sheets on a stack, while others can gather 1000 sheets. Both devices use the same action, but only with a different amount of maximum sheets.

In RSX, this can be solved by copying the action and making the action specific for the device. It would be nice to be able to constrain the properties specifically for a device. This can be done by either parameterizing the action based on the device or by allowing to constrain properties directly on the device.

6.2 Language Usability

Definition of Intent When defining a model in RSX, you specify the definitions of properties, devices, and actions in a single file, alongside the definition of the intent. Although this works fine for demonstrating the language, when using the language in practice this could become inconvenient. The reason is that intent would be specified at a different moment and with different intervals than the specification of properties, devices, and actions. Where

the latter would be defined only once, the intent has to be defined for every individual job. Therefore, it would be more convenient to define intent separately from the other definitions. This can be done by allowing an RSX model to be split into separate files, or by changing the way intent is defined in general. For example, it makes sense to integrate the definition of intent into the existing control software.

Route Optimization By default, RSX generates a model for which only a satisfactory solution is expected. MiniZinc then allows us to generate all satisfiable solutions. However, in some cases, it would be interesting to sort these solutions and find an optimal solution. In RSX, this cannot be defined at the moment, but MiniZinc allows you to minimize or maximize a cost function. A simple optimization would be to minimize the number of actions in a route. However, sometimes it is not as easy to determine the optimal route. For example, sometimes an operator only wants to use a device if no other option is available. Another example is that an operator sometimes only wants to combine two jobs and optimize the total production time of these two jobs combined. It is not trivial for these two examples to determine a cost function and let the constraint solver optimize that function. Further research could be performed to determine what would be a good approach to finding an optimal route in these more difficult cases.

Integration in Workbench RSX is developed in Spoofax. An advantage of this language workbench is that, alongside providing the tools for defining your language, it also automatically generates an IDE for your language. Our implementation of RSX also has an IDE generated by Spoofax, which has syntax highlighting, type checking, and reference resolution. However, a runtime environment in which the generated MiniZinc code is run is missing. This means that the IDE which is generated for RSX can be used to define models and generate MiniZinc code from it, but the user has to manually run the generated MiniZinc code. This can be done either from the command line or by copying and pasting the generated MiniZinc code to the MiniZinc editor. While this approach results in the user being able to find a solution, it makes the feedback loop of developing an RSX model and getting a solution quite long. Furthermore, because the solution is given on the MiniZinc level, the solution needs to be manually mapped back to the level of RSX. We could shorten the feedback loop by integrating the solving step directly in Spoofax. Then, while typing a model in RSX, we could already let the solver run in the background and give immediate feedback when a model is, for example, unsatisfiable. Furthermore, then it is easier to automatically map back a solution given by the solver to the RSX level.

6.3 Relation to CSX

RSX can be seen as an extension of CSX. While the two languages are developed separately, RSX could, in principle, be used for the same purposes as CSX. Both languages can help in finding a valid or optimized parameter configuration, based on intent. However, CSX needs to be given a predefined production route, while RSX can find this route by itself. Although it might seem that RSX is therefore more advanced, it is important to note that the RSX implementation is basic in comparison to CSX. The focus of the development of RSX is more on researching a proof-of-concept for flexible route finding than on developing a usable language for finding a valid parameter configuration. This means that RSX has coverage gaps in the features required for making detailed models, as we show in Section 6.1. For example, in that section, we describe that RSX only supports primitive types, whereas CSX has a much more advanced type system with user-defined composite types, list types, and enumerable types. Also, CSX has the possibility of defining constraints on many levels, where RSX only allows putting in- and output constraints on actions. Furthermore, CSX is

interactive, because the solver is integrated in the language and solutions are automatically mapped back to the level of CSX. In general, CSX can be seen as a much more developed language in terms of usability than RSX.

The question that arises is whether we could make use of both the advanced features of CSX and the flexible route finding of RSX. Two options for this could be:

1. Combine the languages of RSX and CSX into a pipeline, where a route that is found by RSX can be used as input for a configuration model in CSX.

2. Combine the languages of RSX and CSX into one language, which makes use of both the advanced features of CSX and the flexible routing features of RSX.

The first option would require the modeling of a high-level production setup in RSX. Based on this model, a route can be found which serves as a basis for a model in CSX. This CSX model can then be used to get a detailed configuration. Using this approach, there is no need for defining a very detailed model in RSX. This means that it is not required to extend RSX with language features that are already covered by CSX. However, this assumes that a high-level model can be used to find a valid route. In this thesis we only evaluated small cases and determining whether such a high-level representation is sufficient, must be researched further.

The other option of combining the two languages into one does allow the user to model just a single model to find both a route and a configuration based on a single intent. This is nice from a user's perspective because only a single model has to be created, but it requires more work on language development. All language features of CSX need to be transformed in such a way that they can be used in a constraint model where there is no fixed route. Furthermore, the language features of RSX need to be transformed so that they can be used with the more advanced modeling capabilities of CSX. Further research has to be performed to see whether this is possible.

6.4 Threats to Validity

It is common for software engineering research that uses empirical methods to be vulnerable to threats of validity [1]. Our thesis is no exception. Therefore, in this section, we will address these threats one by one.

6.4.1 External Validity

This thesis was written in cooperation with an industrial partner. All our research was done together with this partner and was not validated at other companies. This raises questions about the generalizability of our approach. Would our approach of using a domain-specific language and constraint solving also be possible to implement in other contexts? Our evaluation cannot answer this question. Especially the evaluation of the language coverage is specific to the context of our industrial partner. To counter this threat, we tried to set up our evaluation in such a way that the approach can easily be replicated. The protocol for the evaluation of the language coverage could in theory be repeated in a different context.

6.4.2 Internal Validity

The language coverage evaluation was done by the author of this thesis and a domain expert who was also involved in the development of our approach. This raises the question whether there is a bias in the evaluation of the language. To counter this threat, in the last evaluation

session we also involved two developers of the software that is currently implemented to determine the route in a production plan at our industrial partner. In the future, this evaluation could be repeated with other stakeholders, to ensure that the bias of evaluation is removed.

6.4.3 Construct Validity

To measure the accuracy of RSX, we have written a test suite. These tests can be inaccurate themselves. To counter this, we took time to validate the expectation of each test individually. Furthermore, the benchmarks for the performance tests can be influenced by factors outside the implementation of our language. To counter this threat, we took an average of 10 measurements for each benchmark. Moreover, the benchmarks were run on a device that, apart from the benchmarks, had the operating system running. This device was not connected to a network either.

Chapter 7

Related work

In this chapter, we discuss other work related to our research. We focus on a few works that we encountered during our research. The topics include constraint solving in manufacturing, the use of constraint solving as a backend to a DSL, and the work on CSX.

The use of constraint-solving in manufacturing is not new. The use of an *expert system* for production planning was reviewed by Kusiak and Chen [19]. An expert system is a system that can be consulted to help in making a decision. Moreover, in the area of factory planning, constraint solving is used for finding a factory layout for flexible production systems [16]. Both the use of constraint solving in production planning and factory planning touch on the area of the research in this thesis. However, both cases are not concerned with the routing of an actual production job. Factory planning concerns the planning of a factory layout. Production job scheduling could make use of a generated route, but it only tries to find an efficient time and device allocation for a set of jobs based on their production route and other factors. Factory planning could involve the process of finding the best possible layout, such that certain production routes are possible. However, the actual routing configuration for a job is not something factory planning is involved in.

Another area of manufacturing where constraint solving is generally used, is the area of CAD/CAM software, which stands for Computer-Aided Design (CAD) and Computer-Aided Manufacturing (CAM). Integrating such software with the production properties also requires the solving of configuration parameters and routing sequences [7, 15, 21]. A difference between the integration of CAD/CAM software and RSX is that the number of devices and the variability of devices is different. The integration of CAD/CAM software mostly involves the configuration of paths within a single, constant device. The configuration and pathfinding in this device might not be trivial and requires constraint solving. On the other hand, RSX is designed to find a route which is a combination of multiple devices and paths between these devices.

Constraint solving as a backend for a DSL is also researched in other works. In other areas than manufacturing, languages have been developed which are comparable to RSX in their conceptual design. Keshishzadeh, Mooij, and Mousavi [18] use a constraint-solving backend for a DSL to detect faults early, before compilation. AlleAlle is a language that translates to SMT constraints for defining relational and non-relational models for a wide range of problems [24]. At DATEV, a domain-specific language is developed for efficiently implementing payroll calculations [26]. These can be validated using an integrated SMT solver. All these related works provide an abstraction over a constraint model such that it is easier to express domain-specific concepts as constraints. This is the same approach and idea behind RSX. The difference to RSX is the specific domain and the application of the solver and language.

The closest related work to RSX is the work on CSX [9]. It is already mentioned several times in this thesis and RSX was inspired by it. CSX is a DSL for defining finishers in digital

printing systems. These definitions can then be used for the configuration space exploration of these finisher devices. CSX has many commonalities with RSX. The two languages share the domain of digital printing systems and share the same compilation target language. Furthermore, both languages were developed using Spoofax. The main difference between CSX and RSX is that RSX focuses on routing space exploration, while CSX only supports configuration space exploration based on a predefined route. In Section 6.3 we went deeper into the differences and commonalities between CSX and RSX.

Chapter 8

Conclusion

The work presented in this thesis is a case study into use of domain-specific language using a constraint solving backend to assist in decision-making for the routing problem for digital printing systems as described in Chapter 2. The challenges of this problem are the complexity of finding a solution, the variability of different problem instances, due to the vast variety of finishing devices, and the searchability of the solution space, due to potentially multiple routes being valid. Currently, these challenges are tackled by hard-coding several routes for inline devices, based on common products. However, deviations from these products or these hard-coded routes require manual route finding.

In our case study, we researched a different approach and wanted to answer whether the challenges of this problem can be tackled by a domain-specific language and constraint solving. We did this with the design and implementation of RSX. The main objective for this language was to let a user model realistic domain cases and by doing so, reduce the challenges of solving the routing problem for these cases. Inspired by the earlier research and implementation of CSX, RSX makes use of constraint modeling and a generic constraint solver to achieve this.

We modeled the problem in the constraint programming language MiniZinc first. This model was then used as a basis for how we transform RSX into MiniZinc. We developed RSX in the Spoofax language workbench. In this workbench, we built a syntax and static semantics definition for RSX and transformations for RSX to MiniZinc. From this specification of RSX, we could automatically derive an IDE, containing syntax highlighting, type checking, and name resolution. Automatic generation of MiniZinc code from the program is also part of this IDE. This MiniZinc code has to be manually fed to a constraint solver, which provides us with one or multiple solutions on the level of the MiniZinc code.

We evaluated our implementation on three different evaluation aspects: language coverage, language accuracy, and language performance. In the language coverage evaluation, we conclude that RSX can be used to model realistic instances of the routing problem in the domain of digital printing systems. However, we can only model routes that can be modeled using a linear sequence of steps. Furthermore, the language coverage can be improved by adding composite types. We found that the compilation and solving of these instances can be done in the order of seconds. Moreover, we concluded that the implementation at this moment is accurate for use as a proof of concept of the use of RSX in solving the routing problem.

The implementation of RSX presented in this thesis can be seen as a proof of concept for the use of a DSL in combination with constraint solving for tackling the routing problem in digital printing systems. It shows that the principle of modeling a route as constraints and using this as a basis for a DSL is possible. However, its implementation is not usable in practice at the moment. There are a few aspects of the implementation that could be researched or engineered further to be able to use the language in practice.

One important aspect of using the language in practice is the integration into the workflow of using digital printing systems. The usage of the language is now mainly for showing the capabilities of the language. However, when using it in practice, its usage will most likely be different. We therefore suggest that future research should focus on improving the usability of the language. For example, the language currently requires manually feeding a constraint solver with the MiniZinc model. The results of the constraint model are then expressed in terms of the MiniZinc model. Further research could focus on automating this process and mapping the solution back to high level RSX terms. Furthermore, the specification of devices, actions, and properties should be done at a different time than the specification of intent. Where devices, actions, and properties do not change regularly, the intent will change for each printing job. Currently, both intent and device, action, and property specifications are defined in the same RSX file. It is interesting to research whether intent should indeed be declared at the same time as the more static aspects of the model.

Another aspect we recommend future work should focus on is the combined use of the RSX and CSX languages. Currently, they are two stand-alone languages with different implementations for parsing, static semantics checking, and transformation. However, they are complementary implementations. Where RSX can be used to find a route of devices and specific actions in these devices, CSX can find a valid and optimized parameter configuration for a specific route. Using both these languages to create a full production plan could be the next step. Combining the two languages into one language is also possible. Then, the flexibility in routing of RSX would be added to the CSX language. This would introduce the concepts of pre- and postconditions, device path constraints, and inline device constraints to CSX. The fact that both languages compile into MiniZinc could aid in the integration of the languages.

Overall, we think that by designing and implementing RSX we have shown that it is possible to make use of a domain-specific language to assist in finding a route in digital printing systems. Further research can potentially ensure that this approach is also used in practice by operators in digital printing industry.

Bibliography

[1] Apostolos Ampatzoglou et al. "Identifying, categorizing and mitigating threats to validity in software engineering secondary studies". In: *Information & Software Technology* 106 (2019), pp. 201–230. DOI: 10.1016/j.infsof.2018.10.006. URL: https://doi.org/10.1016/j.infsof.2018.10.006.

[2] Hendrik van Antwerpen et al. "Scopes as types". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018). DOI: 10.1145/3276484. URL: https://doi.org/10.1145/3276484.

[3] Maria J. García de la Banda et al. "The Modelling Language Zinc". In: *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*. Ed. by Frédéric Benhamou. Vol. 4204. Lecture Notes in Computer Science. Springer, 2006, pp. 700–705. ISBN: 3-540-46267-8. DOI: 10.1007/11889205_54. URL: http://dx.doi.org/10.1007/11889205_54.

[4] Clark W. Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability - Second Edition*. Ed. by Armin Biere et al. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 1267–1329. ISBN: 978-1-64368-161-0. DOI: 10.3233/FAIA201017. URL: https://doi.org/10.3233/FAIA201017.

[5] Martin Bravenboer et al. "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of Computer Programming* 72.1-2 (2008), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003. URL: http://dx.doi.org/10.1016/j.scico.2007.11.003.

[6] Broes de Cat et al. "Predicate logic as a modeling language: the IDP system". In: *Declarative Logic Programming: Theory, Systems, and Applications*. Ed. by Michael Kifer and Yanhong Annie Liu. ACM / Morgan & Claypool, 2018, pp. 279–323. ISBN: 978-1-97000-199-0. DOI: 10.1145/3191315.3191321. URL: https://doi.org/10.1145/3191315.3191321.

[7] Tien-Chien Chang and Richard A. Wysk. "Integrating CAD and CAM through automated process planning". In: *International Journal of Production Research* 22.5 (1984), pp. 877–894. DOI: 10.1080/00207548408942506. URL: https://doi.org/10.1080/00207548408942506.

[8] Philippe Codognet and Daniel Diaz. "Compiling Constraints in clp(FD)". In: *Journal of Logic and Algebraic Programming* 27.3 (1996), pp. 185–226.

[9] Jasper Denkers et al. "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*. Ed. by Radu Calinescu and Corina S. Pasareanu. Vol. 13085. Lecture Notes in Computer Science. Springer, 2021, pp. 423–442. ISBN: 978-3-030-92124-8. DOI: 10.1007/978-3-030-92124-8_24. URL: https://doi.org/10.1007/978-3-030-92124-8_24.

[10] Jasper Denkers et al. "Taming Complexity of Industrial Printing Systems Using a Constraint-Based DSL — An Industrial Experience Report". unpublished. 2022.

[11] K. A. Ericsson and H. A. Simon. *Protocol Analysis: Verbal Reports as Data*. MIT Press, 1985.

[12] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: http://www.martinfowler.com/articles/languageWorkbench.html.

[13] Eugene C. Freuder. "In Pursuit of the Holy Grail". In: *ACM Computing Surveys* 28.4es (1996), p. 63.

[14] Alan M. Frisch et al. "The Rules of Constraint Modelling". In: *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*. Ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. Professional Book Center, 2005, pp. 109–116. ISBN: 0938075934. URL: http://www.ijcai.org/papers/1667.pdf.

[15] Tarun Gupta and Biman K Ghosh. "A survey of expert systems in manufacturing and process planning". In: *Computers in Industry* 11.2 (1989), pp. 195–204. DOI: 10.1016/0166-3615(89)90106-1. URL: https://www.sciencedirect.com/science/article/pii/0166361589901061.

[16] Fadil Kallat et al. "Using Component-based Software Synthesis and Constraint Solving to generate Sets of Manufacturing Simulation Models". In: *Procedia CIRP* 93 (2020). 53rd CIRP Conference on Manufacturing Systems 2020, pp. 556–561. DOI: 10.1016/j.procir.2020.03.018. URL: https://www.sciencedirect.com/science/article/pii/S2212827120305783.

[17] Lennart C. L. Kats and Eelco Visser. "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*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497. URL: https://doi.org/10.1145/1869459.1869497.

[18] Sarmen Keshishzadeh, Arjan J. Mooij, and Mohammad Reza Mousavi. "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*. Ed. by Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti. Vol. 8137. Lecture Notes in Computer Science. Springer, 2013, pp. 182–196. ISBN: 978-3-642-40560-0. DOI: 10.1007/978-3-642-40561-7_13. URL: http://dx.doi.org/10.1007/978-3-642-40561-7%5C_13.

[19] Andrew Kusiak and Mingyuan Chen. "Expert systems for planning and scheduling manufacturing systems". In: *European Journal of Operational Research* 34.2 (1988), pp. 113–130. DOI: 10.1016/0377-2217(88)90346-3. URL: https://www.sciencedirect.com/science/article/pii/0377221788903463.

[20] Kim Marriott et al. "The Design of the Zinc Modelling Language". In: *Constraints* 13.3 (2008), pp. 229–267. DOI: 10.1007/s10601-008-9041-4. URL: http://dx.doi.org/10.1007/s10601-008-9041-4.

[21] Huikang K. Miao, Nandakumar Sridharan, and Jami J. Shah. "CAD-CAM integration using machining features". In: *Int. J. Computer Integrated Manufacturing* 15.4 (2002), pp. 296–318. DOI: 10.1080/09511920110077502. URL: http://dx.doi.org/10.1080/09511920110077502.

[22] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. "A nanopass infrastructure for compiler education". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. Ed. by Chris Okasaki and Kathleen Fisher. ACM, 2004, pp. 201–212. ISBN: 1-58113-905-5. DOI: 10.1145/1016850.1016878. URL: http://doi.acm.org/10.1145/1016850.1016878.

[23] Luis Eduardo de Souza Amorim and Eelco Visser. "Multi-purpose Syntax Definition with SDF3". In: *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*. Ed. by Frank S. de Boer and Antonio Cerone. Vol. 12310. Lecture Notes in Computer Science. Springer, 2020, pp. 1–23. ISBN: 978-3-030-58768-0. DOI: 10.1007/978-3-030-58768-0_1. URL: https://doi.org/10.1007/978-3-030-58768-0_1.

[24] Jouke Stoel, Tijs van der Storm, and Jurgen J. Vinju. "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*. Ed. by Hidehiko Masuhara and Tomas Petricek 0001. ACM, 2019, pp. 46–61. ISBN: 978-1-4503-6995-4. DOI: 10.1145/3359591.3359726. URL: https://doi.org/10.1145/3359591.3359726.

[25] Eelco Visser. "WebDSL: A Case Study in Domain-Specific Language Engineering". In: *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*. Ed. by Ralf Lämmel, Joost Visser, and João Saraiva. Vol. 5235. Lecture Notes in Computer Science. Braga, Portugal: Springer, 2007, pp. 291–373. ISBN: 978-3-540-88642-6. DOI: 10.1007/978-3-540-88643-3_7. URL: http://dx.doi.org/10.1007/978-3-540-88643-3_7.

[26] Markus Voelter et al. "A Domain-Specific Language for Payroll Calculations: An Experience Report from DATEV". In: *Domain-Specific Languages in Practice: with JetBrains MPS*. Ed. by Antonio Bucchiarone et al. Springer, 2021, pp. 93–130. ISBN: 978-3-030-73758-0. DOI: 10.1007/978-3-030-73758-0_4. URL: https://doi.org/10.1007/978-3-030-73758-0_4.

[27] Niklaus Wirth. "On the Design of Programming Languages". In: *IFIP Congress*. 1974, pp. 386–393.

[28] Christopher Woods. *Visible Language: Inventions of Writing in the Ancient Middle East and Beyond*. Oriental Institute of the University of Chicago, 2010.

Acronyms

AST abstract syntax tree

CAD computer aided design

CAM computer aided manufacturing

CSX configuration space exploration

DSL domain-specific language

IDE integrated development environment

RSX routing space exploration

Appendix A

Performance Benchmark Results

	Average Runtime (ms)							
Benchmark	**Total**	**Parse**	**Desugar**	**Instantiate**	**Uniquify**	**Explicated**	**MiniZinc AST**	**MiniZinc String**
C1	$0.204 \cdot 10^3$	1.97	0.0167	0.347	0.0523	0.0469	0.683	$0.201 \cdot 10^3$
C2	$0.0808 \cdot 10^3$	0.983	0.00917	0.247	0.0272	0.0262	0.303	$0.0792 \cdot 10^3$
C3	$1.81 \cdot 10^3$	9.14	0.0524	0.353	0.158	0.177	2.63	$1.80 \cdot 10^3$
C4	$1.48 \cdot 10^3$	4.21	0.0345	1.29	0.162	0.181	2.87	$1.47 \cdot 10^3$

Table A.1: Compilation times of the different benchmark cases.

	Average Runtime (ms)	
Benchmark	**Compilation**	**Solving**
S1	$1.81 \cdot 10^3$	298
S2	$1.47 \cdot 10^3$	231

Table A.2: Total compilation and solving times of the different benchmark cases.

Appendix B

RSX Grammar

ID	::=	`[a-zA-Z][_a-zA-Z0-9]*`	Identifier
model	::=	$[\![definition]\!]^*$	Model
definition	::=	**property** ID **:** $type$	Property
	|	**inline** $ID[\![$ **->** $ID]\!]^*$	Inline
	|	**device** ID **{** $[\![path]\!]^*$ **}**	Device
	|	**action** ID **{** $[\![member]\!]^*$ **}**	Action
	|	**action** ID **(** $[\![param[\![$**,**$param]\!]^*]\!]^?$ **) {** $[\![member]\!]^*$ **}**	Action with parameters
	|	**intent** exp	Intent
path	::=	**path** $[\![ID[\![$ **->** $ID]\!]^*]\!]^?$	Path
member	::=	**constraint** $expr$	Constraint
	|	**preserves** ID	Preserve
param	::=	ID **:** $type$	Action parameter
type	::=	**int**	Integer type
	|	**nat**	Natural number type
	|	**bool**	Boolean type
	|	**unit**	Unit type
expr	::=	**true**	True
	|	**false**	False
	|	i	Integer
	|	ID	Variable
	|	ID**.in**	Variable in
	|	ID**.out**	Variable out
	|	e**.present**	Present
	|	e**.absent**	Absent
	|	$expr$ **and** $expr$	And
	|	$expr$ **or** $expr$	Or
	|	$expr$ **==** $expr$	Equal
	|	$expr$ **!=** $expr$	Not equal
	|	$expr$ **<** $expr$	Less
	|	$expr$ **>** $expr$	Greater
	|	$expr$ **<=** $expr$	Less or equal
	|	$expr$ **>=** $expr$	Greater or equal
	|	$expr$ **+** $expr$	Addition
	|	$expr$ **-** $expr$	Subtraction
	|	$expr$ ***** $expr$	Multiplication
	|	$expr$ **/** $expr$	Division
	|	$expr$ **%** $expr$	Modulo
	|	**not** $expr$	Boolean negation
	|	**-**$expr$	Numeric negation
	|	**(** $expr$ **)**	Brackets

Figure B.1: Grammar of RSX.

Appendix C

RSX Typing Rules

C.1 Environments

The typing rules make use of four different environments. This first environment is denoted by A, which holds the set of defined actions in the RSX model. The second environment is D, which holds the set of defined devices in the RSX model. The third environment is P, which is only set when evaluating the members of an action. It holds a mapping of all parameters of the action by their name. The last environment is E, which holds a mapping of all property types by their name.

When we declare an action with parameters we denote the mapping of parameters in P as $P[p_x/T_x]$, which means that in that action, the parameter p_x has type T_x and therefore $P(p_x) = T_x$ holds. In cases where no parameters can be assigned, we denote the absence of the P environment using the $\perp$ symbol.

C.2 Rules

The typing rules of RSX are described in Figure C.1, Figure C.2, Figure C.3 and Figure C.4.

$$\frac{E(ID) = T}{A, D, \perp, E \vdash \texttt{property}\ ID : T}\ \textsc{Property} \qquad \frac{A, D, \perp, E \vdash e : \mathit{bool}}{A, D, \perp, E \vdash \texttt{intent}\ e}\ \textsc{Intent}$$

Figure C.1: Typing rules of properties in RSX.

$$\frac{\begin{array}{c} n \geqslant 1 \\ d_1 \in D \qquad d_2 \in D \qquad \ldots \qquad d_n \in D \end{array}}{A, D, \perp, E \vdash \texttt{inline}\ d_1\ \texttt{->}\ d_2\ \texttt{->}\ \ldots\ \texttt{->}\ d_n}\ \textsc{Inline}$$

$$\frac{\begin{array}{c} ID \in D \\ A, D, \perp, E \vdash p_1 \\ A, D, \perp, E \vdash p_2 \\ \vdots \\ A, D, \perp, E \vdash p_n \\ n \geqslant 0 \end{array}}{A, D, \perp, E \vdash \texttt{device}\ ID\ \texttt{\{}\ p_1, p_2, \ldots, p_n\ \texttt{\}}}\ \textsc{Device}$$

$$\frac{\begin{array}{c} n \geqslant 0 \\ a_1 \in A \qquad a_2 \in A \qquad \ldots \qquad a_n \in A \end{array}}{A, D, \perp, E \vdash \texttt{path \{}\ a_1\ \texttt{->}\ a_2\ \texttt{->}\ \ldots\ \texttt{->}\ a_n\ \texttt{\}}}\ \textsc{Path}$$

Figure C.2: Typing rules of devices in RSX.

$$\frac{\begin{array}{c} ID \in A \\ A, D, \bot, E \vdash m_1 \\ A, D, \bot, E \vdash m_2 \\ \vdots \\ A, D, \bot, E \vdash m_n \\ n \geqslant 0 \end{array}}{A, D, \bot, E \vdash \texttt{action}\ ID\ \texttt{\{}\ m_1, m_2, \ldots, m_n\ \texttt{\}}}\ \textsc{Action}$$

$$\frac{\begin{array}{c} ID \in A \\ A, D, P, E \vdash m_1 \\ A, D, P, E \vdash m_2 \\ \vdots \\ A, D, P, E \vdash m_n \\ n \geqslant 0 \qquad k \geqslant 0 \\ P[p_1/T_1][p_2/T_2] \ldots [p_k/T_k] \end{array}}{A, D, \bot, E \vdash \texttt{action}\ ID\texttt{(}\ p_1 : T_1, p_2 : T_2, \ldots, p_m : T_k\ \texttt{)\{}\ m_1, m_2, \ldots, m_n\ \texttt{\}}}\ \textsc{Action-WithParams}$$

$$\frac{A, D, P, E \vdash e : \mathit{bool}}{A, D, P, E \vdash \texttt{constraint}\ e}\ \textsc{Constraint} \qquad \frac{E(ID) = T}{A, D, P, E \vdash \texttt{preserve}\ ID}\ \textsc{Preserve}$$

Figure C.3: Typing rules of actions in RSX.

$$\frac{}{A, D, P, E \vdash \texttt{true} : bool}\ \textsc{Bool-True} \qquad \frac{}{A, D, P, E \vdash \texttt{false} : bool}\ \textsc{Bool-False}$$

$$\frac{i \text{ is an integer literal}}{A, D, P, E \vdash i : int}\ \textsc{Int} \qquad \frac{P(ID) = T}{A, D, P, E \vdash ID : T}\ \textsc{Param-Read}$$

$$\frac{E(ID) = T}{A, D, P, E \vdash ID\texttt{.in} : T}\ \textsc{Property-In} \qquad \frac{E(ID) = T}{A, D, P, E \vdash ID\texttt{.out} : T}\ \textsc{Property-Out}$$

$$\frac{A, D, P, E \vdash e : T}{A, D, P, E \vdash e\texttt{.present} : bool}\ \textsc{Property-Present}$$

$$\frac{A, D, P, E \vdash e : T}{A, D, P, E \vdash e\texttt{.absent} : bool}\ \textsc{Property-Absent} \qquad \frac{\begin{array}{c}A, D, P, E \vdash e1 : bool \\ A, D, P, E \vdash e2 : bool \\ \bowtie \in \{\texttt{and}, \texttt{or}, \texttt{==}, \texttt{!=}\}\end{array}}{A, D, P, E \vdash e1 \bowtie e2 : bool}\ \textsc{Bool-Compare}$$

$$\frac{\begin{array}{c}A, D, P, E \vdash e1 : int \\ A, D, P, E \vdash e2 : int \\ \bowtie \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{==}, \texttt{!=}\}\end{array}}{A, D, P, E \vdash e1 \bowtie e2 : bool}\ \textsc{Int-Compare} \qquad \frac{\begin{array}{c}A, D, P, E \vdash e1 : int \\ A, D, P, E \vdash e2 : int \\ \bowtie \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}\end{array}}{A, D, P, E \vdash e1 \bowtie e2 : int}\ \textsc{Int-Arith}$$

$$\frac{A, D, P, E \vdash e : bool}{A, D, P, E \vdash \texttt{not}\ e : bool}\ \textsc{Bool-Neg} \qquad \frac{A, D, P, E \vdash e : int}{A, D, P, E \vdash \texttt{-}e : int}\ \textsc{Int-Neg}$$

Figure C.4: Typing rules of expressions in RSX.