

Domain-Specific Languages for Digital Printing Systems

Jasper Denkers



Domain-Specific Languages for Digital Printing Systems

DISSERTATION

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus Prof.dr.ir. T.H.J.J. van der Hagen;
Chair of the Board for Doctorates
to be defended publicly on
Tuesday 1 October 2024 at 15:00 o'clock

by

Jasper DENKERS

Master of Science in Computer Science,
Delft University of Technology, the Netherlands
born in Rijnwoude, the Netherlands

This dissertation has been approved by the promotor.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof.dr. A.E. Zaidman	Delft University of Technology, promotor
Prof.dr. J.J. Vinju	Eindhoven University of Technology, promotor

Independent members:

Prof.dr. F.W. Vaandrager	Radboud University
Prof.dr. S. Erdweg	Johannes Gutenberg University Mainz
Dr. N. Yorke-Smith	Delft University of Technology
Prof.dr. A. van Deursen	Delft University of Technology
Prof.dr. M.M. de Weerdt	Delft University of Technology, reserve member

Other member:

Dr. L. C. M. van Gool	Canon Production Printing
-----------------------	---------------------------

Prof.dr. E. Visser (Delft University of Technology) was the original promotor and supervisor of this research until his untimely passing on April 5th, 2022.

The work in this dissertation has been carried out at the Delft University of Technology and at Canon Production Printing, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

This research was supported by a grant from the Top Consortia for Knowledge and Innovation (TKIs) of the Dutch Ministry of Economic Affairs and by Canon Production Printing.



Copyright © 2024 Jasper Denkers – <https://www.jasperdenkers.nl>

Cover image: Copyright © Atchariya63 / Adobe Stock

IPA Dissertation Series: 2024-13

Printed by: Gildeprint – <https://www.gildeprint.nl>

ISBN: 978-94-6496-197-3

Contents

Summary	vii
Samenvatting	ix
Preface	xi
1 Introduction	1
1.1 Domain-Specific Languages	4
1.1.1 Domain-Specific Software Languages	5
1.1.2 Advantages of DSLs	6
1.1.3 Disadvantages of DSLs	7
1.1.4 Comparison to Other Software Engineering Approaches	7
1.1.5 DSLs in Industry	9
1.2 Language Workbenches	10
1.3 DSLs at Canon Production Printing	11
1.4 Research Objective	12
1.5 Research Method	13
1.6 Case Studies & Contributions	14
1.6.1 CSX: Configuration Space eXploration	14
1.6.2 OIL: Open Interaction Language	16
1.7 Structure & Origin of Chapters	17
2 CSX 1.0: Configuration Space Exploration for Digital Printing Systems	19
2.1 Introduction	20
2.2 Finishers in the Digital Printing Domain	22
2.2.1 Perfect Binding	23
2.3 CSX	24
2.3.1 Configurations and Jobs	27
2.3.2 Exploration and Validation	28
2.4 Denotational Semantics	28
2.5 Implementation	31
2.6 Evaluation	34
2.7 Related Work	35
2.8 Conclusions	37
2.9 Appendix: Declarative Semantics	38
2.10 Appendix: Inhabitation	40
3 CSX 2.0: Taming Complexity of Industrial Printing Systems Using a Constraint-Based DSL — An Industrial Experience Report	43
3.1 Introduction	44
3.2 Industrial Printing and Finishing Systems	47

3.2.1	Printing and Finishing	47
3.2.2	Requirements	50
3.2.3	CSX: <u>C</u> onfiguration <u>S</u> pace <u>e</u> Xploration	50
3.2.4	Coverage Gaps	54
3.3	Increasing Domain Coverage	55
3.3.1	Non-Uniform Stacks of Sheets	55
3.3.2	Geometrical Constructs	58
3.3.3	Functional-Style Operators	61
3.4	Industrial Evaluation	63
3.4.1	Domain Coverage	64
3.4.2	Configuration Accuracy	77
3.4.3	Configuration Performance	78
3.4.4	Relevance	80
3.5	Discussion	83
3.5.1	Language Design	83
3.5.2	Constraint-Based Programming	88
3.5.3	Application in Practice	90
3.5.4	Lessons Learned	92
3.5.5	Threats to Validity	92
3.6	Related Work	93
3.7	Conclusions	95
3.7.1	Future work.	95
4	OIL: an Industrial Case Study in Language Engineering with Spoofox	97
4.1	Introduction	98
4.2	Spoofox	100
4.2.1	Anatomy of Spoofox Projects	101
4.2.2	Data Representation with ATerms	102
4.2.3	Syntax Definition with SDF ₃	102
4.2.4	Static Semantics with NaBL ₂	103
4.2.5	Transformation with Stratego	106
4.2.6	Editor Services with ESV	108
4.2.7	Testing with SPT	108
4.3	OIL	109
4.3.1	History of OIL	109
4.3.2	Overview of OIL	109
4.3.3	Implementation Features	114
4.4	Case Study Context and Method	115
4.4.1	Context	115
4.4.2	Research Method	116
4.4.3	Setup	117
4.5	Concrete Syntax	118
4.5.1	From XML to Custom Syntax	118
4.5.2	Composed Grammars and Disambiguation	121
4.5.3	The Python Implementation	123
4.5.4	Evaluation	125

4.6	Abstract Syntax	129
4.6.1	Intermediate Representations	129
4.6.2	Desugaring	130
4.6.3	Resilient Staging	132
4.6.4	The Python Implementation	133
4.6.5	Evaluation	134
4.7	Static Semantics	139
4.7.1	Well-formedness Checking	139
4.7.2	Cross-file and Cross-language Analysis	140
4.7.3	The Python Implementation	141
4.7.4	Evaluation	142
4.8	Dynamic Semantics	146
4.8.1	Division into Projects	146
4.8.2	Using Static Analysis Results	148
4.8.3	Configurability of the mCRL2 Generator	150
4.8.4	The Python Implementation	151
4.8.5	Evaluation	152
4.9	Evaluation	155
4.9.1	Summary	155
4.9.2	Threats to Validity	156
4.10	Discussion	160
4.10.1	Spoofax’s Strengths	160
4.10.2	Spoofax’s Weaknesses	161
4.10.3	Lessons Learned	163
4.10.4	Spoofax Engineering Agenda	164
4.11	Related Work	166
4.12	Conclusions	168
5	Conclusion	171
5.1	Research Questions	171
5.2	Lessons Learned	173
5.3	Implications & Future Work	175
	Bibliography	177
	Curriculum Vitae	189
	List of Publications	191

Summary

Tools used in software engineering often balance a tradeoff between generality and specificity. The most important tools in software engineering are programming languages, and the most common ones are *General-Purpose Languages (GPLs)*. Because of their generality, GPLs can be used to develop many kinds of software. *Domain-Specific Languages (DSLs)* are a more specific counterpart; they are programming languages tailored to a specific domain. DSLs are not generally applicable but can be more effective for developing software within their particular domain.

DSLs can be beneficial if their benefits outweigh the investments. Advantages attributed to DSLs include improved software engineering productivity, improved automation possibilities, and enabling non-programmer domain experts to contribute to the software engineering process. Disadvantages attributed to DSLs include the costs of introducing and maintaining a DSL and dependency on skills. In practice, it is hard to predict whether a DSL will be beneficial.

Language workbenches are tools for developing and deploying DSLs. They aim to reduce the investment that is required for DSLs and to improve the usability of the created DSLs. By lowering the investment, language workbenches can improve the opportunity for DSLs to be effective. Although much academic work has been published about the underlying technology and concepts of language workbenches, there is little empirical evidence on the actual impact of language workbenches in practice.

In this dissertation, we contribute such empirical evidence on the creation and evaluation of DSLs that are developed with language workbenches. We do so by conducting case studies in an industrial setting. This is important, as such empirical evidence can help others to determine whether to adopt DSLs developed with language workbenches. In particular, we use and evaluate Spoofox, a language workbench developed at the Delft University of Technology.

The context of our work is Canon Production Printing, a digital printing systems manufacturing company. Canon Production Printing provides a good environment for evaluating DSLs as they have obtained extensive domain knowledge for complex domains like modeling behavior, performance, and physical aspects of printing systems. We develop and evaluate DSLs for two of such domains. First, we develop CSX, a new DSL for the domain of configuration spaces of digital printing systems. Second, we reimplement OIL, an existing DSL for control software based on state machines. In both cases, we compare the newly created DSL with the existing situation.

For both cases, we draw generally positive conclusions. For example, in the CSX project, the DSL enables the use of constraint-solving technology which aids automatic and accurate configuration of printing systems, which

can ultimately improve the quality, performance, and usability of printing systems. In the OIL project, we found that Spoofox is more than adequate for developing a complex DSL with industrial requirements and we found indications that it is more productive to develop a DSL with Spoofox compared to using a GPL and available libraries.

Our extensive case studies at Canon Production Printing have taught us valuable lessons and insights. In particular, to make good on the promise of DSLs in industry, language workbenches need to improve in terms of the non-functional aspects. We expect that improving on, e.g., portability, usability, and documentation will improve the impact of Spoofox on industrial DSL development.

Samenvatting

Gereedschap voor softwareontwikkeling balanceert vaak tussen algemeenheid en specificiteit. Het belangrijkste gereedschap voor softwareontwikkeling zijn programmeertalen, en het meest voorkomend zijn *General-Purpose Languages (GPLs)*. Vanwege hun algemeenheid kunnen GPLs worden gebruikt om allerlei soorten software te ontwikkelen. *Domain-Specific Languages (DSLs)* zijn een meer specifieke tegenhanger; dat zijn programmeertalen op maat gemaakt voor een specifiek domein. DSLs zijn niet breed toepasbaar, maar kunnen effectiever zijn voor het ontwikkelen van software binnen hun specifieke domein.

DSLs kunnen voordelig zijn als hun voordelen groter zijn dan de investeringen. Voordelen die aan DSLs worden toegeschreven zijn onder andere verbeterde productiviteit voor softwareontwikkeling, verbeterde automatiseringsmogelijkheden en het mogelijk maken dat domein experts die geen programmeur zijn kunnen bijdragen aan de softwareontwikkeling. Nadelen die aan DSLs worden toegeschreven zijn onder andere de kosten van het introduceren en onderhouden van een DSL en de afhankelijkheid van vaardigheden. In de praktijk is het moeilijk te voorspellen of een DSL voordelig zal zijn.

Language workbenches zijn gereedschap voor het ontwikkelen en implementeren van DSLs. Ze hebben tot doel om de investering die nodig is voor DSLs te verminderen en de gebruiksvriendelijkheid van de gecreëerde DSLs te verbeteren. Door de investering te verlagen, kunnen language workbenches de kans vergroten dat DSLs effectief zijn. Hoewel er veel academisch werk is gepubliceerd over de onderliggende technologie en concepten van language workbenches, is er weinig empirisch bewijs over de daadwerkelijk impact van language workbenches in de praktijk.

In dit proefschrift dragen we dergelijk empirisch bewijs bij over de creatie en evaluatie van DSLs die zijn ontwikkeld met language workbenches. Dit doen we door case studies uit te voeren in een industriële omgeving. Dit is belangrijk, omdat dergelijk empirisch bewijs anderen kan helpen te bepalen of ze DSLs ontwikkeld met language workbenches willen toepassen. In het bijzonder gebruiken en evalueren we Spoofox, een language workbench ontwikkeld aan de Technische Universiteit Delft.

De context van ons werk is Canon Production Printing, een bedrijf dat digitale printsystemen produceert. Canon Production Printing biedt een goede omgeving voor het evalueren van DSLs, omdat ze uitgebreide domeinkennis hebben opgebouwd voor complexe domeinen zoals het modelleren van gedrag, prestaties en fysieke aspecten van printsystemen. We ontwikkelen en evalueren DSLs voor twee van dergelijke domeinen. Ten eerste ontwikkelen we CSX, een nieuwe DSL voor het domein van configuratieruimten van digitale printsystemen. Ten tweede herimplementeren we OIL, een bestaande DSL voor besturingssoftware op basis van state machines. In beide gevallen vergelijken we de nieuw gecreëerde DSL met de bestaande situatie.

Voor beide gevallen trekken we over het algemeen positieve conclusies. Zo maakt de DSL in het CSX project het gebruik van constraint-oplossingstechnologie mogelijk, wat helpt bij automatische en nauwkeurige configuratie van printsystemen, wat uiteindelijk de kwaliteit, prestaties en gebruiksvriendelijkheid van printsystemen kan verbeteren. In het OIL project vonden we dat Spoofox ruimschoots geschikt is voor het ontwikkelen van een complexe DSL met industriële vereisten en we vonden aanwijzingen dat het productiever is om een DSL te ontwikkelen met Spoofox in vergelijking met het gebruik van een GPL en beschikbare libraries.

Onze uitgebreide case studies bij Canon Production Printing hebben ons waardevolle lessen en inzichten opgeleverd. Om de belofte van DSLs in de praktijk waar te maken, moeten language workbenches met name verbeteren op het gebied van niet-functionele aspecten. We verwachten dat verbeteringen op het gebied van portabiliteit, gebruiksvriendelijkheid en documentatie de impact van Spoofox op de ontwikkeling van industriële DSLs zal verbeteren.

Preface

I had three motivations for doing a PhD: taking a deep dive into and contributing to a subject close to my heart, working with smart and inspiring people, and taking on a big challenge. Both fortunately and unfortunately, my expectations have been exceeded on all three fronts. This made the last couple of years an adventure. I am glad the adventure is now over, but I definitely would not have wanted to miss it. I consider it a great privilege to have had the opportunity to pursue a PhD with the people I had around me.

The build-up to this adventure started 20 years ago when my father introduced me to the world of computers. After he gave me the first push, the internet helped me to explore so many interesting things. Starting to sense and appreciate the beauty of programming more and more, I realized programming languages were fascinating pieces of technology.

It was not until my studies in Delft that I began to truly understand where this beauty came from. Eelco Visser taught me the concepts of programming languages and inspired me to explore programming in new ways. This opened a world of possibilities that had a huge impact on me, and eventually shaped my professional life, including starting a PhD with him. His untimely passing marks a somber chapter in my PhD journey. It was a privilege to work with him, and I deeply wish he were here to see this dissertation.

ACKNOWLEDGEMENTS

Although only my name is on the cover of this dissertation, many others have contributed, both directly and indirectly.

Without Eelco, I would not have pursued a PhD. I am thankful to him for inspiring me and giving me the opportunity. Doing a PhD is also about learning about yourself, and Eelco helped me do so. He helped me realize I tend to think about a solution without really understanding the problem first. When communicating about your work, especially when writing a scientific paper, this is problematic. Also, he taught me when to ask for help. Patiently, he made me aware of my pitfalls. He taught me how to write scientific papers and how to do science in general.

For my second promotor, I did not need another expert in programming languages, but someone I knew I could trust and rely on, and preferably someone experienced with doing research in industry. Andy Zaidman was my top choice. While initially he was a bit surprised when I asked him — because of his field of expertise differing from programming languages — I am very happy he joined my team. I am thankful to Andy for all our nice interactions and his strong support when writing papers. I very much appreciate his leadership style. There have been times I was stuck and I was procrastinating, lacking motivation. Andy often sensed these moments and got me back on track. With very clear words, but always in the most friendly and respectful

way. If I will get into leadership positions, I will think of his example.

When Eelco passed away, who could fill the gap in my team? I could not think of someone better than Jurgen Vinju. During our collaboration, I realized that his relationship with Eelco was even more valuable than I knew before, nicely characterized by Jurgen as them being academic brothers. I am thankful to Jurgen for stepping into a role that I can imagine is emotional for him as well. Jurgen was essential in architecting my two most important papers. Without Jurgen's help, I do not know how these papers would have ever reached the finish line. In addition to his deep analytical skills — something I had only experienced before when working with Eelco — I very much appreciate the enthusiasm and positive energy he consistently brings. He inspires me to keep having fun and to never stop programming.

Doing a PhD within a company sounds nice, but without strong support from a champion at the company, the research can easily fail. I am very happy that Louis van Gool was my champion. Louis brings a unique blend of a strong theoretical and academic background, combined with practical and industrial experience. I am thankful to Louis for providing a great environment to do research, minimizing the need for me to navigate corporate processes. Louis maintained a high standard and actively contributed to papers through reviews with a great level of detail, which I very much appreciated.

After my first presentation to a large audience at Canon, where I shared the results of my first months of work, Marvin Brunner approached me with an idea that could be interesting to have a look at. I am glad he did, as this idea eventually led to the biggest project of my PhD, the CSX project. My research heavily relies on extensive domain knowledge accumulated over years within a company, and Marvin was great at transferring this knowledge. I am thankful to Marvin for often asking "Hoe zit dit nou eigenlijk echt?" ("What is really going on here?"), pushing us to better understand the problem we were working on. Companies should be very happy with employees like Louis and Marvin.

A large part of my work was in close collaboration with Olav Bunte. It was nice to both do our PhD during the same period, even though he started later and still managed to overtake me. I think our complementary skills made us a great team, especially when it came to writing papers. The highlight of our collaboration was our last journal paper on the OIL project, which was a very tough one to write. With a large set of co-authors, we had to navigate with many captains on our ship. So many times we thought we were on the right track, only to find ourselves needing to go back to the drawing board. I am thankful to Olav for his perseverance and attention to detail. I am proud that we got the paper published, together.

At Canon Production Printing, many others have contributed to my work. Several Master's students did their thesis projects with us: Mark Frenken, Tom Buskens, Samuël Noah Voogd, Bram van Walraven and Jordi van Laarhoven. I appreciated the results those students achieved in a short amount of time. I am thankful to all other employees who contributed to OIL and CSX, through participation in our workshops, taking the time to share domain knowledge, or

through nice discussions on language engineering in general. Finally, I want to thank Jack van der Elsen for his guidance and support in filing our patent related to CSX.

I am grateful to my colleagues in Delft, particularly those in the Programming Languages group, for their welcoming and supportive attitude. I very much appreciate the dedication with which so many members of the group have contributed to Spoofox, alongside their numerous research and teaching responsibilities. What they, along with previous generations of PhD students, have achieved with Spoofox is truly impressive. I am also thankful to Maarten Sijm for choosing to do his thesis project on parsing with us — it was a lot of fun working together.

I want to thank the members of my PhD committee for taking the time to evaluate my dissertation: Frits Vaandrager, Sebastian Erdweg, Neil Yorke-Smith, Arie van Deursen, and Mathijs de Weerd. I also want to thank Neil for teaching a course on constraint programming at the Institute for Programming research and Algorithmics (IPA) in 2019. Rarely have I learned so much in a single day. Neil's course gave me a kickstart on what would later become a fundamental part of the CSX project.

I am fortunate to have many friends who have supported me along the way. I want to thank a few in particular. I am thankful to Mike Loef, for always being there for me. As someone who listened to so many of my complaints, I am proud of him for still starting a PhD. I am thankful to my paranimphs, Hessel de Gelder and Eloy Testerink, for supporting me during the defense. Hessel and I held weekly accountability sessions, which helped a lot in learning about my pitfalls and staying focused on the right things. Eloy, with his Monday morning music suggestions, often provided encouragement to start the week on a positive note.

I want to thank my family. In particular, I am thankful to my parents, for their unconditional support and for giving me the freedom and opportunity to pursue the things that I wanted in life. Finally, I am deeply thankful to the love of my life, Manon, for her support and patience. I know it has not always been easy to stand by my side, but I am proud of what we have achieved together over the past few years. I very much look forward to sharing the rest of our lives together.

Introduction

1

The ever-increasing role of software in society is fueled by the programming languages in which we write software. In daily life, we use natural languages such as English to communicate with each other. For communication about instructing computers, we use programming languages. With this communication, we develop and maintain software by expressing instructions for a computer to solve problems we care about. It is essential that these instructions are expressed and interpreted accurately, such that the computer can deliver the correct outcomes for the computations the software was intended for. It is also essential that these instructions are understandable by anyone contributing to software because, otherwise, miscommunication arises in the contributions and the quality of the outcomes of the software is in danger. In this dissertation, we investigate how the development and application of *Domain-Specific Languages (DSLs)* [1, 2, 3] impact the expression and interpretation of instructions for computers. DSLs are programming languages that are tailored to a specific domain. In particular, we do our investigation in the context of Canon Production Printing¹, a digital printing systems manufacturing company.

Before further discussing DSLs, we discuss their more commonly used counterpart: *General-Purpose programming Languages (GPLs)*. Software engineers often instruct computers by writing code in GPLs. Examples of GPLs are C++, Java, and Python, which can be used for writing software in many domains such as games, webshops, and medical data analysis. GPLs provide building blocks (i.e., abstractions, notations, or constructs) that allow programmers to instruct computers without being bothered with low-level machine code. Still, developing software that controls printing systems using GPLs remains challenging. For example, the functionality of the printing systems could be improved if the control software would support finding optimal configurations, but due to the complexity of the printing systems, this is hard to achieve.

DSLs provide building blocks that are typically larger than the building blocks in GPLs, and that are specific to the DSL's domain. These building blocks thus allow the expression of instructions directly in terms of the domain [1]. This prevents programmers from repeating low-level technical details and boilerplate code each time a new problem is solved using the DSL. Potentially, it also enables domain experts who are not programmers to write or validate software code [2]. A typical characteristic is that the code written in a DSL is translated automatically to an existing language such as a general purpose one [3]. Although using a DSL comes at the cost of investing in the development and maintenance of the DSL itself, its benefits could advance software development in complex industrial contexts such as at Canon Production Printing.

¹<https://cpp.canon>

```

1 device ExamplePerfectBinder {
2     location bookIn : Stack
3     location coverIn : Sheet
4
5     // Max dimensions of input
6     [1000 ≤ bookIn.height and bookIn.height ≤ 3000]
7     [2000 ≤ bookIn.width and bookIn.width ≤ 5000]
8
9     component toMill = ToMill(bookIn, milledBook) {
10        [millingDepth ≤ 30] // Max 3mm of milling
11        [bookIn.thickness < 170] // Max 17mm book thickness
12    }
13
14    location milledBook : Stack
15
16    component toCrease = ToCrease(coverIn, creasedCover) {
17        [minDist ≥ 50] // At least 5mm between creases
18    }
19
20    location creasedCover : CreasedSheet
21
22    component toCover = ToCover(creasedCover, milledBook, out) {}
23
24    location out : PerfectBoundBook
25 }

```

Figure 1.1: An example of code written in a DSL. The code models a device that performs perfect binding, a binding technique that produces books by gluing a cover around a stack of sheets. The DSL in which the code is written is CSX, developed in this dissertation. CSX is used to generate software that can automatically configure printing systems.

As an example, we discuss a DSL called CSX that is developed in this dissertation for the Configuration Space eXploration of digital printing systems. Configuration space exploration is the process of finding a configuration that specifies the complete manufacturing process (the input materials, the device parameters, and the end product). Challenges in realizing configuration space exploration are making it automatic, accurate, and optimal. Automation means that, given a description of an intended product by an operator, the software automatically finds a valid configuration. Accuracy concerns that the delivered configurations conform to the device’s capabilities and constraints. Ideally, configuration space exploration supports finding optimal configurations, e.g., to configure devices to maximize print productivity or minimize paper waste.

Figure 1.1 shows an example of code written in CSX. The code models a perfect binder. A perfect binder is a finishing device, i.e., it processes sheets of paper after printing. This device uses the perfect binding technique, which means that it produces books by gluing a cover sheet around a stack of sheets². The code captures physical limitations of the device, e.g., the maximum dimensions of the input materials or the maximum amount of milling³ that can occur; concerns relevant for the configuration of the device. The `location` keyword (see, e.g., lines 2 and 3), which is used to represent the physical locations of the devices, is an example of how CSX supports the expression of (physical) aspects of the device directly in terms of the domain of printing systems.

CSX enables and automates the use of SMT solving [4]: technology that can automatically find an (optimal) solution among many possibilities. This is relevant for the configuration of printing systems because they can be configured in many ways. SMT solving can overcome the challenges of realizing configuration space exploration. However, the code required to instruct constraint solving is low-level, hard to understand, and tedious to maintain. CSX tackles these problems by automatically generating the code necessary for constraint solving. Therefore, in the CSX project, one of the key advantages of a DSL is automation through code generation. This enables the integration of useful technology that ultimately can improve the quality of printing systems. CSX will be further discussed in Chapter 2 and Chapter 3.

DSLs have been investigated extensively in research and have been applied in practice frequently for over 20 years [1, 5]. DSLs have been attributed improvements to the software engineering process in terms of productivity, quality, maintainability, and taming accidental complexity [1, 2]. Still, little is known about the impact in practice of DSLs developed with state-of-the-art tooling. Although Voelter et al. demonstrated that systems of significant size can be developed in industrial settings using a particular tool, they call for more studies on the evaluation of such tooling [6]. This is necessary for testing whether language workbenches are suitable for developing software in a wide range of domains.

This research, which was done in close collaboration with our industrial

²The printed version of this dissertation is produced using the perfect binding technique.

³Milling makes the paper along the spine rough, improving the adherence of the glue.

partner Canon Production Printing, created a unique opportunity to evaluate the application of DSLs. The key question of this dissertation is how DSLs and related tooling can impact software engineering within our particular industrial context. In particular, we investigated the creation of DSLs using *language workbenches* [7, 8]. Language workbenches provide the tools and infrastructure for developing and deploying DSLs with accessible usage environments. In our investigation, we compare DSLs developed with a language workbench to existing solutions developed with traditional software engineering techniques.

We investigate this question by performing case studies of DSL development at Canon Production Printing for projects that have already been running for several years, for which the domains have been analyzed extensively, and for which software components already exist. This setup enables us to take a deep dive into idiosyncratic industrial cases that can produce lessons that neither a generic theory nor a large empirical survey can produce. The outcomes may be harder to generalize, but they are also more valid because we can manually take into account the possible noise factors. Threats to internal validity are easier to mitigate, although threats to external validity (generalizability) shall remain. Using this method, we made contributions in the following categories:

DSL Creation We (re-)design and implement DSLs using a language workbench from the viewpoint of replacing existing software components.

Industrial Evaluation We evaluate the newly created DSLs and compare them with the existing situation.

Lessons Learned We collect lessons learned for DSL development in an industrial setting.

In the remainder of this chapter, we cover background information on domain-specific languages, language workbenches, and our industrial context. Then, we describe our research method and introduce the case studies. Finally, we summarize our contributions and describe the structure of this dissertation.

1.1 DOMAIN-SPECIFIC LANGUAGES

We previously referred to programming languages as a means for communicating about software, and how such communication can be domain-specific. Domain-specific communication is present outside the software context as well. Aircraft marshallers use visual signaling expressed using their arms to guide aircraft on the ground. The visual signs are clear and concise — tailored to the domain of guiding aircraft — and easy to interpret for a pilot. DSLs aim to bring the same advantages — expressing instructions clearly and concisely — to software engineering.

In this section, we define DSLs in the context of software and we discuss advantages, disadvantages, comparisons to other software engineering approaches, and their use in industry.

1.1.1 Domain-Specific Software Languages

The following definition of domain-specific (software) languages was proposed by Van Deursen et al. [1]:

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

From this definition, we derive the following key characteristics of DSLs:

Programming Language Like GPLs, DSLs are programming languages, and thus are used to write software that can be executed by computers.

Domain-Specificity DSLs are focused on and restricted to a particular domain.

Focused Expressive Power DSLs provide building blocks specific to the DSL's domain, which can be textual, graphical, or a combination thereof.

The example of visual signaling by aircraft marshallers has the characteristics of domain-specificity and focused expressive power. However, it is not a programming language that can be executed by computers. From now on, we will use the term DSL to refer to domain-specific languages in the context of software, conforming to the above definition. We will refer to the process of developing and maintaining DSLs as *language engineering* and to people doing this as *language engineers*.

In the domain-specificity characteristic, it remains vague what kinds of domains a DSL can be specific to. We use the following definition of Czarnecki and Eisenecker [9]:

A domain is an area of knowledge scoped to maximize the satisfaction of the requirements, including a set of concepts and terminology understood by practitioners in the area, and including knowledge of how to build software systems (or their parts) in the area.

DSLs can be classified across several dimensions. We discuss two of such dimensions to indicate the scope of this dissertation.

Internal vs. External DSLs [7]. Internal (or embedded) DSLs are developed, deployed, and used within an existing GPL. This GPL is then called the host language. By making creative use of the host language's syntax, the user of an internal DSL can express domain-specific building blocks within the host language. On the other hand, external (or standalone) DSLs are languages with a custom syntax and usage environment. Typically, the investment for developing an internal DSL is lower than for an external DSL [2]. However, the development of an internal DSL is restricted by the host language's syntax and usage environment [2]. In this dissertation, we focus on external DSLs. An external DSL developed with traditional techniques was already present at Canon Production Printing, which enables us to compare this DSL implementation with a re-implementation using a language workbench.

Horizontal vs. Vertical DSLs [10]. "Horizontal" and "vertical" refer to the organization of software typically consisting of layers and columns. The

horizontal layers are of a technical nature, corresponding to, e.g., low-level machine code or a high-level web application framework. The vertical columns relate to application or business domains. Horizontal DSLs thus target a technical domain. Horizontal DSLs are typically used by software engineers only and aim to improve writing software for a particular technical task. Examples of horizontal DSLs are SQL for the domain of database querying and CSS for styling websites. On the other hand, vertical DSLs target an application or business domain and might have non-programmer domain experts as users. Traditionally, domain experts involved in software engineering write requirements in, e.g., Word documents that software engineers interpret and write software for. This gap between requirements and software engineering often involves miscommunication and thus negatively impacts the process, which Fowler calls the “worst bottleneck in software development” [2]. Vertical DSLs potentially bridge this gap by enabling domain experts to write or read code in a DSL. For example, the Risma DSL enabled financial experts to validate the correctness of interest rate products [11]. In this dissertation, we consider both a horizontal and a vertical DSL.

1.1.2 Advantages of DSLs

DSLs are popular for several reasons [2]. The following advantages attributed to DSLs motivated their use in our case studies at Canon Production Printing:

Facilitating Automation & Integration Using code generation, DSLs can generate code for (automation of) integration with other technology [3]. For example, this improves the accessibility of technologies requiring low-level code such as constraint solving or model checking. This is relevant as enabling the use of such technology can improve the quality of printing systems.

Involving Domain Experts Fowler considers improving communication with domain experts to be one of the two main reasons for DSL’s popularity [2]. Due to the domain-specificity and simplicity compared to GPLs, domain experts can read code written in a DSL or even write code themselves using a DSL. Reading DSL code by domain experts improves validatibility; if domain experts can understand the code and validate it in the DSL’s environment, validation can occur early in the software deployment cycle. Writing or modifying DSL code by domain experts bridges the gap from requirements to implementation, which can overcome the communication overhead between domain experts and programmers. At Canon Production Printing, this could enable, e.g., mechanical engineers to contribute more directly to software engineering.

Improving Productivity Fowler considers improving productivity for developers to be the other main reason for DSL’s popularity [2]. By abstracting over, e.g., boilerplate code and algorithmic details, using a DSL can improve software engineering productivity compared to not using a DSL. Since boilerplate code and algorithmic details are generated automatically, they do not require to be written repeatedly by hand, decreasing programming effort and thereby improving productivity. Improving productivity is relevant in an industrial setting as it can decrease the cost of software engineering.

Validation and Optimization Because DSLs are at the abstraction level of the DSL's domain, they allow for automated domain-specific validation and optimization [1], reducing errors and improving quality. Possibly, validation already occurs in the IDE, while editing programs written in the DSL, and can prevent errors early in the development process. This is relevant for Canon Production Printing as it could shorten the development cycle and improve the quality of the products.

Other advantages attributed to DSLs include [1, 12, 3]: code written in DSLs is concise, self-documenting, and reusable; DSLs improve quality, maintainability, reusability, reliability, traceability, and portability; DSLs capture domain knowledge enabling the conservation and reuse of this knowledge; DSLs improve testability; and DSLs enable simulation, model checking, and verification.

1.1.3 *Disadvantages of DSLs*

We consider the following disadvantages attributed to DSLs to be relevant in the context of our case studies:

Costs: Introduction and Maintenance There is a cost to introducing and maintaining a DSL [1]. Next to the software language engineering component of introducing and maintaining a DSL, both language engineers creating the DSL as well as users of the DSL need to be trained [2]. These costs should be affordable.

(Unclear) Return on Investment Both the investment and outcome of introducing a DSL depend on many factors, which makes it a risk that introducing a DSL is not beneficial, i.e., that it has no positive return on investment [3]. This risk makes it difficult to opt for using a DSL in an industrial setting.

Learning Curve Developing and applying DSLs in practice is non-trivial due to the required language engineering skills [1]. There is a steep learning curve for software engineers to learn language engineering [13], especially for developing external DSLs, which can be a hurdle for adopting DSLs in an industrial setting.

Dependency on Tooling Introducing a DSL with, e.g., a language workbench, introduces a dependency on an external party, although this dependency can be smaller if the language is open source. In an industrial setting, such a dependency could conflict with corporate policies.

Other disadvantages attributed DSLs to include [1]: limited availability; difficulty in finding the proper scope for a DSL; the difficulty of balancing between specificity and generality; and potential loss of efficiency in generated code compared to manually optimized code.

1.1.4 *Comparison to Other Software Engineering Approaches*

We compare DSLs with other software engineering approaches, mostly by discussing abstraction. Abstraction involves the collection of information concerning a specific purpose, and the hiding of information that is not relevant

for that purpose [3]. Using abstraction therefore simplifies information processing, and thus it is often used in software engineering to tackle complexity. We refer to the *level of abstraction* as the dimension that indicates how much simplification and hiding of complexity occurs. Finding a suitable level of abstraction is challenging, as hiding complexity is an advantage on the one hand, but on the other hand, a too high level of abstraction might oversimplify. How abstraction plays a role in software varies per software engineering method. We discuss the role of abstractions in GPLs, model-driven software engineering, and low-code/no-code software engineering, and compare these approaches to DSLs.

GPLs. GPLs themselves are already an abstraction compared to low-level machine code. When using a GPL like Java, you use keywords and symbols and a compiler takes care of translating the GPL code to low-level machine code a computer can execute. Still, as a software engineer, you need to handle all kinds of technical aspects such as memory management or concurrency. Abstractions provided by the GPL can be used to ease the handling of such aspects. For example, *subroutines* can be used to define a reusable piece of code that can be called repeatedly without repeating the details of its implementation. A software *library* is a collection of abstractions such as subroutines, typically for some domain, that can be imported and reused across software projects. For example, a software library could provide abstractions for processing images. Similarly, *frameworks* implemented in a GPL can be used to implement a particular type of application. For example, a framework for developing web applications provides infrastructure for, e.g., a server-client architecture, handling browser requests, and validating user input through web forms. The most important difference between GPLs and DSLs is that DSLs are more tailored to a specific domain, rather than being general, and that DSLs are at a higher level of abstraction. The higher level of abstraction and domain-specificity of a DSL enables its users to benefit from the earlier discussed advantages but also limits the scope in which it can be applied. DSLs and GPLs also differ in terms of user experience. First, the previously discussed abstractions within GPLs (subroutines, libraries, and frameworks) require an understanding of and experience with programming. Typically, a DSL also contains abstractions, but, in contrast to GPLs, it has limited support for adding new abstractions. Although adding new abstractions is powerful, it is hard to understand for non-programmers [14], and thus restricting the possibilities for creating new abstractions within DSLs can likely make them more accessible. Second, using a GPL comes with its usage environment tailored to programmers, which is typically not accessible to non-programmers. A DSL allows for introducing a usage environment that is tailored to the specific domain, which can make them more accessible to non-programmers.

Model-driven software engineering (MDSE). Models are used in many engineering disciplines. For example, mechanical engineers use three-dimensional models of bikes. Such models are abstractions of reality made with a given purpose in mind [3] such as design or production. MDSE [15] refers to the

method of software engineering that focuses on using models and modeling as opposed to programs and programming [3]. However, modeling and programming can be closely related [16], and models can also be captured using code. Based on such models, software gets automatically generated. In MDSE, the models capture properties that are relevant to the realization of software. There are several ways of giving shape to MDSE, e.g., by configuring the parameters of a fixed model, modeling using a general-purpose modeling language, drawing a model in a visual DSL, or writing models in a textual DSL. Textual DSLs are a linguistic viewpoint on MDSE and thus one of the design patterns for implementing MDSE. Although many interpretations of MDSE and its variants exist, MDSE and DSLs both are often used for the accurate capturing of information on a higher level of abstraction, that hides complexity and automates the realization of software. Typically, this is motivated with the aim of improving the software engineering process, e.g., improving productivity or involving (non-programmer) domain experts. Canon Production Printing uses MDSE with DSLs in various engineering disciplines for their development of digital printing systems [17].

Low-code/no-code (LCNC). Low-code/no-code is a software engineering method that can be used to create software requiring little to no code [18], relying on MDSE [3]. It focuses on improving software engineering productivity and enabling non-programmers to contribute to software engineering. Typically, LCNC involves visual abstractions offered through graphical user interfaces, which allows non-programmers to develop software. In contrast to DSLs, we could see LCNC as a user interface configuration viewpoint on MDSE; it provides a visual user interface in which (a model of) an application can be configured. LCNC solutions are typically tailored to realizing a particular type of applications such as web applications, thereby addressing technical concerns similar to horizontal DSLs. Compared to vertical DSLs, LCNC solutions are more general and can thus lack the advantages that vertical DSLs bring due to their domain-specificity. Similar to DSLs, LCNC can involve (non-technical) business experts to speed up the software engineering process. LCNC can be more cost-effective compared to DSLs as it does not require the investment of creating a DSL.

1.1.5 DSLs in Industry

Although DSLs have the potential to solve relevant problems, they come with significant development and maintenance costs. When using DSLs in practice, the question is whether their use will lead to a positive return on investment. Van Deursen and Klinkt state that a successful application of DSLs in industry should strike a proper balance between the benefits and risks [1]. Mernik et al. state that deciding whether to adopt a DSL should be substantiated by concerns such as improving software economics (the benefits should outweigh the costs) and enabling software engineering by domain experts with little or no software engineering experience [5].

Voelter gives three considerations for determining whether it is suitable to use a DSL [19], although it remains unclear how to evaluate for these criteria:

(1) the DSL should address a suitable domain that features richness and variety, involve software that requires boilerplate or derived artifacts, or a need for platform independence; (2) the organization adopting the DSL should have a suitable business model that has a sufficient outlook for the DSL to return on its investment, that could benefit from improved software engineering effectivity, or that targets a focused domain; and (3) the organization adopting the DSL should have enough technical maturity to manage the development cycle of a DSL and it should have a willingness and ability to change.

1.2 LANGUAGE WORKBENCHES

A language workbench is “an environment designed to help people create new DSLs, together with high-quality tooling required to use those DSLs effectively” [2]. They prevent “reinventing the wheel” while developing DSLs and their accompanying environments (IDEs), and thereby promise to reduce the cost of introducing and maintaining DSLs. Examples of language workbenches are Spoofox [20], MPS [21], Xtext [22], Rascal [23], MontiCore [24], Gemoc [25], and Neverlang [26]. Language workbenches vary, e.g., in supported notations (textual, graphical, or projectional) and types of maintainers (academic or industrial). An overview and comparison of language workbenches is given by Erdweg et al. [27, 8].

Developing a DSL for a particular domain involves coping with the complexity inherent to that domain. The “ideal” language workbench should enable language engineers to focus as much as possible on that inherent domain complexity and on finding a proper language design and a design for the usage environment. On the other hand, a language engineer should focus as little as possible on practical aspects such as implementing the usage environment and deployment. Language workbenches that fulfill this ideal could stimulate DSL adoption by amplifying a DSL’s advantages and reducing its disadvantages. This stems from the hypothesis that if we can reduce the cost of developing a DSL, it can also improve their opportunity for a positive return on investment. Also, deploying a DSL with a user-friendly environment makes a DSL more accessible to its intended users.

Although many language workbenches promise to support the development of DSLs close to the ideal, and there is ample literature on the underlying theory of language workbenches (e.g., [8, 28]), little is documented about the actual added value of language workbenches in practice or about their perceived usefulness [29]. In addition to the supposed advantages of language workbenches, they also come with disadvantages. For example, using a language workbench introduces a dependency on an external tool, possibly with vendor lock-in [12]. Also, adopting a language workbench comes with a steep learning curve [30] and thus education costs.

Voelter et al. reported on an extensive case study using the MPS language workbench under non-trivial industrial requirements [6]. Although they found meaningful lessons learned for the particular case and language workbench, the authors call for more studies on language workbench evaluation to expand our knowledge of the usefulness of language workbenches in general. This is

important as it will help industrial language engineers and companies decide when and how to use language workbenches for DSL development.

The questions raised above motivated us to investigate the introduction of DSLs in an industrial context using a state-of-the-art language workbench. In particular, we do so with the Spoofox language workbench⁴.

Spoofox is an open-source language workbench for which it was promised to support the development of textual DSLs by offering meta-DSLs (DSLs for developing DSLs) for concise, declarative specifications of languages and usage environments [31]. The idea of declarative language specification is that language developers can think as much as possible about the high-level features of their languages rather than focusing on the low-level implementation of, e.g., parsing or type-checking algorithms. Based on language aspect specifications in the meta-DSLs, Spoofox automatically generates a usage environment. Further technical details of Spoofox will be discussed in Section 4.2.

Spoofox originates from decades of research [32] and many of its underlying components and theory have been evaluated. Typically, these evaluations involve small or artificial cases. An exception to this is Visser’s case study on the development of WebDSL [33], which discusses language design and implementation for a DSL in the domain of web programming. The development of WebDSL has been continued by Groenewegen, including application and validation of WebDSL by creating and deploying real-world web applications [34]. However, this work does not evaluate Spoofox itself, let alone evaluate it in an industrial context. We hypothesize that if the industrial evaluation of our research increases confidence that Spoofox will bring significant improvements in practice, that could positively impact its adoption opportunities. If we learn that improvements to Spoofox are still needed, then that would be valuable lessons for the language engineering community.

1.3 DSLS AT CANON PRODUCTION PRINTING

Our research has been conducted at Canon Production Printing, a large digital printing systems manufacturing company. Manufacturing digital printing systems involves the cooperation between several engineering disciplines: physics, chemistry, electrical-, mechanical-, and software engineering. The interplay between these disciplines and the strict quality and performance requirements imposed on the printers make their development complex. To tackle complexity, many of these engineering disciplines use models. The company’s software research and development department has a long history of applying model-driven software engineering (MDSE). More recently, the company also uses language workbenches to develop DSLs for coping with, e.g., variability, encoding mechanical properties, and hardware-software interfaces [17].

While early results are promising, the company struggles to scale the adoption of the technology. Training engineers to become language engineers is hard. The language engineers that develop DSLs feel like having to “reinvent the wheel” for particular language engineering tasks like code generation

⁴<https://spoofox.dev>

because best practices and clear DSL design patterns are lacking. It is unclear to what extent tool characteristics influence the success of introducing DSLs. To use DSLs for critical parts of printers that directly impact the quality of the systems, a high level of confidence in the language engineering techniques and tools is needed. Despite the widely recognized potential of DSLs and language workbenches, the above problems still make investing in DSL technology in industry risky.

Most of the language engineering activities at Canon Production Printing have been conducted with the MPS language workbench. In an overview of some of the applications of MPS at Canon Production Printing, Schindler et al. conclude "Moving to a model-based way-of-working is mostly a no-brainer at Canon Production Printing. However, choosing MPS as core technology to bridge domain-specific interpretation gaps certainly is not." [17], which is due to several challenges experienced with MPS. By using Spoofox as the language workbench in our research, we create the opportunity to evaluate a different tool within the same context. Spoofox mostly differs from MPS in the sense that it uses textual notation rather than projectional notation and that it is developed by an academic party rather than a commercial one.

The environment at Canon Production Printing provides a unique opportunity to study DSLs developed with Spoofox. By applying the model-driven approach in many projects, Canon Production Printing obtained extensive domain knowledge for complex domains like modeling behavior, performance, and physical aspects of printing systems. The domain analysis outcomes of these projects are valuable assets for experimenting with DSL development. Additionally, the environment has the potential for a big impact by DSL solutions. An example of where a DSL solution could have a big impact is the integration of printers with finishing devices such as booklet makers. The integration of such finishing devices, some of which are produced by external companies, with Canon Production Printing's printers has a significant software component and it is currently an expensive process. Therefore, there are interesting opportunities to experiment with the usage of DSLs as a means to integrate externally produced devices with the software from Canon Production Printing, potentially even with external companies as users of such DSLs.

1.4 RESEARCH OBJECTIVE

The underlying goal of this research is to improve our understanding of how DSLs developed with language workbenches can impact industrial software engineering. This leads to our main research question:

RQ-Main: *How do DSLs developed with a state-of-the-art language workbench impact industrial software engineering?*

Rather than taking a broad perspective, we investigate a specific language workbench (Spoofox) within a specific industrial context (Canon Production Printing). We aim to investigate this particular combination deeply, ultimately

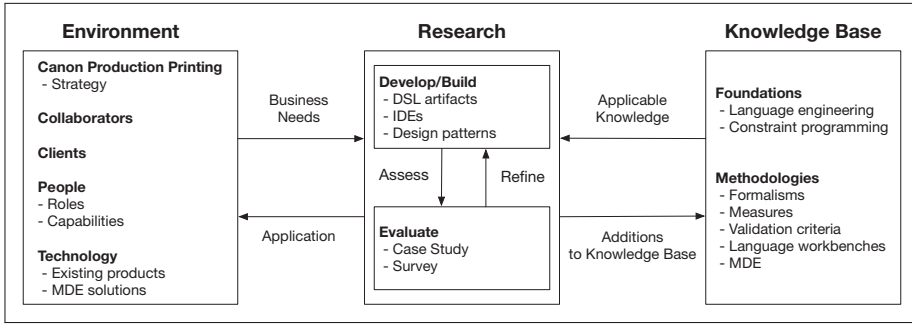


Figure 1.2: Our instantiation of Hevner et al.’s design science framework [35].

contributing to our general understanding of DSLs developed with language workbenches. We do so by conducting two case studies of DSL development, as further described in Section 1.5 and 1.6.

For our first case study (CSX), we investigate the following research question:

RQ-CSX: *How does a DSL with a constraint-solving backend impact the development of control software for digital printing systems with respect to domain coverage, accuracy, and performance?*

For our second case study (OIL), we investigate the following research question:

RQ-OIL: *How does the productivity of implementing an industrial language in Spoofox compare to the productivity when using a GPL and available libraries?*

1.5 RESEARCH METHOD

Our overarching research method is based on Hevner’s design science framework [35]. Figure 1.2 depicts our instantiation of this framework. It incorporates behavioral aspects, relevant for doing research in an industrial environment. The proposed research builds on existing foundations and methodologies for developing DSLs. In addition, it takes the business needs that arise from the environment into account. Given both sources, we performed several iterations of case studies of developing and evaluating DSL artifacts. The artifacts are applied at Canon Production Printing and thus have to contribute to the industrial environment. The outcomes of these case studies contribute back to the knowledge base. We further discuss the three components of our instantiation of the framework below.

Environment. Canon Production Printing is characterized by the development of high-end products. This is part of the company’s strategy and implies the need for the development of advanced software. There are multiple categories of potential DSL users: Canon Production Printing’s employees, external companies producing devices that interface with Canon Production Printing

printers, and clients that configure their systems. Last, an important part of the environment is the software for existing products and existing solutions that are used. These aspects define the problem space in which we perform our research.

Knowledge Base. We build on a knowledge base consisting of foundations and methodologies relevant to our work. In particular, we apply existing language engineering techniques and tools. In addition, we experiment with combinations of technologies. For example, a DSL with constraint solving as a backend enables declarative notations and fuses DSL technology with constraint programming.

Research. Our research concerns the development and evaluation of DSL implementations at Canon Production Printing. We use the Spoofox language workbench [20, 31] for the development of the DSLs. This limits the scope of the tooling perspective and allows us to focus on the evaluation of a particular type of language workbench. For Spoofox, important characteristics are that the notation is textual and the maintaining party is a research group.

We select projects based on the following criteria:

1. **Existing Domain Analysis and Implementation** The projects have extensive domain analysis available and have an existing implementation using GPLs.
2. **No Use of Language Workbench** The projects do not already use a DSL, or if they do use a DSL, it is not developed using a language workbench.
3. **Complexity** The projects should involve complexity which makes the engineers working on it struggle with advancing their solutions using the existing approach.

1.6 CASE STUDIES & CONTRIBUTIONS

We now summarize the core contributions of this dissertation which are centered around two case studies of DSL creation that we have conducted at Canon Production Printing. For each case study, we will describe the domain and corresponding software engineering problem it concerns, indicate how the case conforms to our selection criteria, and why using a DSL developed with a language workbench is relevant. Furthermore, we list the contributions that we make in each case study.

1.6.1 CSX: Configuration Space eXploration

The CSX case study concerns the development of a DSL for the domain of configuration space exploration of printing systems. Configuration space exploration is the process of finding a valid configuration that specifies the complete manufacturing process (the input materials, the device parameters, and the end product). Canon Production Printing develops flexible printing systems that are highly complex systems that consist of printers, that print individual sheets of paper, and finishing equipment, that processes sheets after printing, for example, assembling a book. Integrating finishing equipment with printers involves the development of control software that configures the

devices, taking hardware constraints into account. This control software is highly complex to realize due to (1) the intertwined nature of printing and finishing, (2) the large variety of print products and production options for a given product, and (3) the large range of finishers produced by different vendors.

The CSX case conforms to our selection criteria because: (1) there is an existing implementation of control software using traditional technologies that captures decades of domain analysis, (2) no external DSL has been developed for developing such control software, and (3) due to the complexity of control software, engineers struggle to develop correct and complete control software.

In the CSX project, we develop a domain-specific language that offers an interface to constraint solving specific to the printing domain. We use it to model printing and finishing devices and to automatically derive constraint solver-based environments for automatic configuration. In addition to realizing configuration space exploration that is automatic and complete, it also supports finding optimal configurations. For example, this can be used to configure booklet makers in a way that maximizes printing productivity or minimizes paper waste. CSX could be seen as both a horizontal and a vertical DSL. CSX is a horizontal DSL in the sense that it is an abstraction over the technical domain of constraint solving. CSX is a vertical DSL as it covers the business domain of the configuration spaces of printing devices.

Using CSX is motivated in the following ways. First, CSX realizes configuration space exploration that is automatic and complete (i.e., covers all possible configurations), whereas the existing situation involved the implementation of device-specific heuristics that were not necessarily complete. Although constraint solving is a natural fit for developing control software that is automatic, modeling a printing system – including all the details of the mechanics – in a generic constraint modeling language is tedious. Using a DSL makes it accessible to use constraint solving in the backend, and to support finding optimal configurations. Second, CSX can help cope with the large variety of printing systems; the DSL can be used to model the unique characteristics of devices while the DSL itself captures the commonalities. We evaluate the impact of CSX concerning these two motivations. Furthermore, using CSX is motivated in ways for which evaluation is outside the context of this dissertation. Using CSX could enable mechanical engineers to contribute to the modeling process and thereby reduce the communication overhead between mechanical- and software engineers. Also, the use of a DSL could improve software engineering productivity by reducing the turnaround time required for developing control software.

Contributions. Related to the CSX case study (see Chapter 2 and Chapter 3), we make the following contributions, labeled by our previously introduced categories of contributions:

- **DSL Creation:** CSX 1.0, a declarative language for the specification of finishers at the conceptual level of the domain, including:
 - a declarative semantics of CSX 1.0,

- a denotational semantics of CSX 1.0 in terms of SMT constraints, and
- a programming environment for CSX 1.0 that integrates SMT solving as a backend.
- **Industrial Evaluation:** An evaluation of CSX 1.0 based on two printing system cases from Canon Production Printing (a perfect binder and a booklet maker).
- **DSL Creation:** CSX 2.0, an extension of CSX 1.0, which extends the language’s coverage of the printing domain by adding support for generic lists, geometrical constructs, and functional-style operators.
- **Industrial Evaluation:** An evaluation of the domain coverage, accuracy, performance, and relevance of CSX 2.0 in the context of Canon Production Printing.
- **Lessons Learned:** Lessons learned on using a constraint-based DSL in an industrial context.

1.6.2 OIL: Open Interaction Language

The OIL case study concerns the re-implementation of an existing DSL using Spoofox. OIL is a DSL for modeling control software based on state machines, developed internally at Canon Production Printing as no existing solution was found suitable [17]. OIL is a horizontal DSL, as it covers the technical domain of control software, not a business domain. In contrast to the CSX project that compares a new approach using a DSL with the existing approach without a DSL, the OIL project compares two implementations of the same DSL. For OIL a custom design and implementation of the language itself using XML syntax and Python already existed. Although the custom implementation of OIL led to a working solution, its development involved reinventing the wheel and the solution lacks features typically provided by language workbenches such as editor support.

The OIL case conforms to our selection criteria because: (1) there is an existing custom implementation of the OIL language using XML and Python and existing components have been replaced by implementations using OIL, (2) the existing implementation of OIL does not use a language workbench, and (3) OIL and the domain it covers are sufficiently complex that this complexity hinders further development.

By re-implementing OIL in Spoofox, we obtain two implementations of the same DSL: one with custom technologies and one using Spoofox. This enables us to make a fair comparison and get an indication of the effect of using a language workbench. In particular, we evaluate the productivity of using Spoofox compared to not using a language workbench. This is relevant for two main reasons. On the one hand, there is opportunity: there exist DSL implementations in industry that have not been developed with the potential benefit of language workbenches. On the other hand, there are still unknowns: language workbenches spawned from academic environments can have different views on software engineering effectiveness compared to a pure

industrial setting.

Contributions. Related to the OIL case study (see Chapter 4), we make the following contributions:

- **DSL Creation:** An implementation of the OIL language in the Spoofox language workbench.
- **Industrial Evaluation:** An evaluation of whether Spoofox’s original claims — on making language development, compared to not using a language workbench, more productive — stand when realizing the implementation of a complex industrial language such as OIL.
- **Lessons Learned:** Strengths, weaknesses, and an agenda for future engineering on Spoofox.

1.7 STRUCTURE & ORIGIN OF CHAPTERS

We now discuss the rest of the structure of this dissertation. The main chapters of this dissertation are based on three peer-reviewed accepted papers and one pending publication. The author of this dissertation is the main contributor and first author of three publications [36, 37, 38], and is shared first author (with equal contributions) with Olav Bunte for the publication on OIL [39].

Since the chapters covering CSX are based on standalone publications with distinct contributions, there is some redundancy in these chapters. However, we chose not to remove that redundancy, to ensure that each chapter can be read independently. The main chapters and their corresponding publication are as follows:

- Chapter 2 (CSX 1.0) is an extended version of the paper *Configuration Space Exploration for Digital Printing Systems* [37] from 2021, published in the proceedings of the *Software Engineering and Formal Methods (SEFM)* conference. Part of the work in this chapter is recorded in the European patent EP3855304 A1⁵, published on 28 July 2021.
- Chapter 3 (CSX 2.0) corresponds to the paper *Taming Complexity of Industrial Printing Systems Using a Constraint-Based DSL — An Industrial Experience Report* [38] from 2023, published in the *Software: Practice and Experience (SPE)* journal.
- Chapter 4 (OIL) corresponds to the paper *OIL: an Industrial Case Study in Language Engineering with Spoofox* [39], published in the journal on *Software and Systems Modeling (SoSyM)*. This chapter incorporates the paper *Migrating Custom DSL Implementations to a Language Workbench (Tool Demo)* [36] from 2018, published in the proceedings of the *Software Language Engineering (SLE)* conference.

Finally, in Chapter 5, we conclude by summarizing our results and we provide directions for future work.

⁵<https://worldwide.espacenet.com/patent/search?q=pn%3DEP3855304A1>

CSX 1.0: Configuration Space Exploration for Digital Printing Systems

2

ABSTRACT

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

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

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

Based on: Jasper Denkers, Marvin Brunner, Louis van Gool, and Eelco Visser. "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*. Vol. 13085. Lecture Notes in Computer Science. Springer, 2021, pp. 423-442. DOI: 10.1007/978-3-030-92124-8_24.

2.1 INTRODUCTION

Digital printing systems are flexible manufacturing systems, i.e., manufacturing systems that are capable of adjusting their abilities to manufacture different types and quantities of products, without expensive hardware changes. The variety in printing applications stems from both printing (printing on sheets of paper) and finishing (processing collections of printed sheets, e.g., to form a book). The *configuration space* for a digital printing system consists of all possible configurations given the system's features and hardware constraints. For producing a booklet of a particular size, a printed stack of sheets can be stitched, it can be folded, and it can be trimmed. Optionally, the sheets can be rotated in an intermediate production step such that a single trimming component can be used for trimming in multiple dimensions. The decisions made for these manufacturing parameters influence important factors such as productivity (production time increases when sheets are rotated) or efficiency (paper is wasted when input sheets are trimmed).

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

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

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

The global characteristics of finishers make the use of constraint (SMT) solving a natural fit for realizing environments for configuration space exploration. For example, trimming the paper along a certain dimension might

impose a specific orientation or transformation in an earlier production step. A constraint-based approach considers its specifications as global and will take into account interdependent system-level constraints when finding solutions, i.e., configurations. A constraint-based model of a finisher contains a representation of the input materials at intermediate locations in the system. However, for modeling domain objects such as sheets and stacks, abstraction mechanisms such as classes are not naturally available in SMT modeling. An SMT model of a finisher requires low-level encoding of the properties of the materials at all locations. Therefore, expressing finishers in SMT by hand is tedious, error-prone, and not in terms of domain concepts. Additionally, an SMT model of a finisher is complex to understand and difficult to maintain.

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

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

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

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

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

- We have developed CSX, a declarative language for the specification of finishers at the conceptual level of the domain. We interpret CSX specifications for several modes of configuration space exploration: checking whether

configurations are valid, finding optimal configurations under objectives, and interactively validating specifications.

- We define a denotational semantics of CSX in terms of SMT constraints that serves as an interface to solvers that can be used to find models in order to check inhabitation of a specification and to explore the configuration space of the specified finisher.
- We realize a programming environment for CSX that integrates an SMT solver as back-end and that presents solutions in terms of the specification.
- We evaluate CSX by specifying two types of finishers: a perfect binder and a booklet maker. For these cases, we benchmark the performance for a configuration space exploration scenario with and without optimization.

2.2 FINISHERS IN THE DIGITAL PRINTING DOMAIN

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

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

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

Ideally, operators can use the combination of a printer with finishers as an end-to-end solution instead of having to configure each device separately. Additionally, optimization capabilities are also useful when considering the system as a whole. For example, an operator would like to produce booklets with the available resources while minimizing paper waste or optimizing

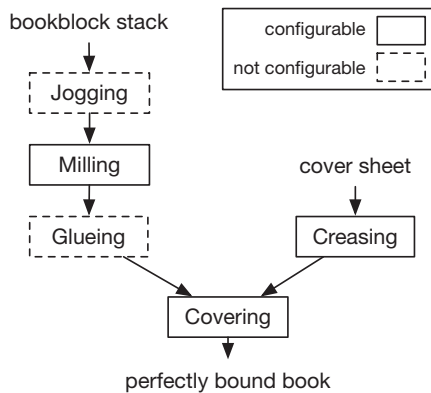


Figure 2.1: Schematic view of the perfect binding book-producing process. Only milling, creasing, and covering are configurable and therefore impact the configuration space. Jogging and glueing are automatically configured by the device itself.

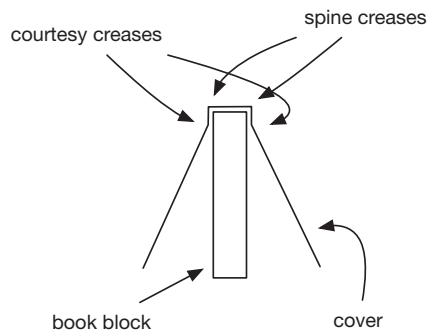


Figure 2.2: A perfectly bound book viewed from the top. Spine creases result into a sharper fold, reduce wrinkles, and improve the fit of the cover around the book block. Courtesy creases ease opening the front and back part of the cover. Glue in the spine holds the book block sheets and cover together.

productivity. If the different configuration possibilities impose a tradeoff between, e.g., resource consumption and productivity, an operator should be able to make a motivated choice with ease, i.e., without thinking about and manually trying out many combinations of configuration parameters.

2.2.1 Perfect Binding

As an example, we discuss a *perfect binder*: a finisher that produces books by binding a stack of sheets with glue and by covering the book block in a cover sheet. A perfect binder typically has two inputs: one for the stack of sheets that form the book block and one for the cover sheets. Figure 2.1 shows the perfect binding process. Figure 2.2 depicts the components of a perfectly bound book, viewed from above.

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

Separately, cover sheets are prepared before being bound around the book block. The preparation consists of creasing, i.e., applying pressure on the paper to ease folding of the paper later. Two creases are applied at the location of the cover that end up along the edges of the spine of the book. These creases improve the fit of the cover along the spine of the book block, supporting a tight fold around the spine. Additionally, two courtesy creases are applied on

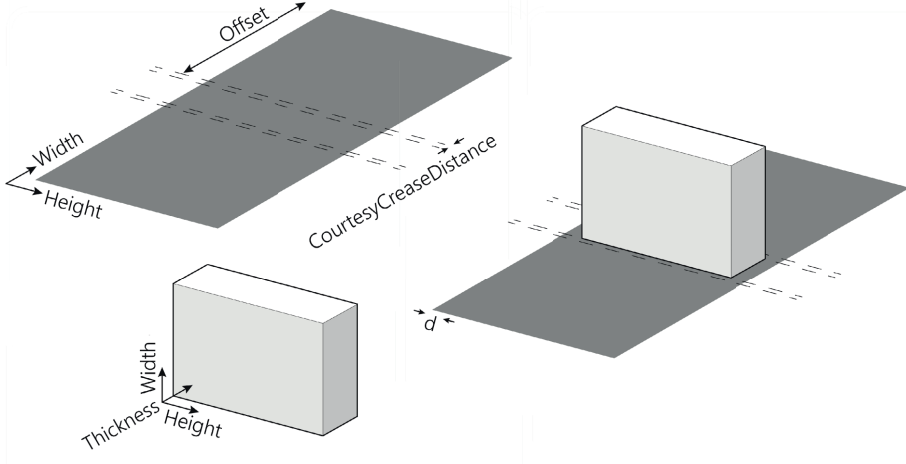


Figure 2.3: Components of a perfectly bound book (cover and book block) and the dimensions as how we use them in the CSX specification.

the cover. Courtesy creases are applied on the front and back of the resulting book to support the folding of the cover sheet. Note that courtesy creases are applied at the opposite side as the spine creases, as they are used for folds in opposite directions.

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

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

2.3 CSX

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


```

1 type Sheet {
2   width: int, [width > 0],
3   height: int, [height > 0]
4 }
5 type Stack {
6   width: int, [width > 0],
7   height: int, [height > 0],
8   thickness: int, [thickness > 0],
9   volume =
10    width * height * thickness
11 }
12 type PerfectBoundBook {
13   book: Stack,
14   frontCover: Sheet,
15   backCover: Sheet
16 }

```

```

1 type CreasedSheet {
2   sheet: Sheet,
3   spineFront: Crease,
4   spineBack: Crease,
5   courtesyFront: Crease,
6   courtesyBack: Crease
7 }
8
9 type Crease {
10  // Offset:
11  off: int,
12  [off ≥ 0],
13  // Direction: 0 = down, 1 = up
14  dir: int,
15  [dir == 0 or dir == 1]
16 }

```

Figure 2.4: The specification of types for the example perfect binder in CSX. Dimensions are in 0.1mm.

configuration space for a given job.

CSX is declarative: a specification in the language describes the behavior and configuration spaces of finishers. A CSX specification does not describe algorithms to compute configurations. Specifications include relations between objects at locations in the systems. We use the language to model devices as sequences of components that perform actions. Components instantiate generic, reusable actions. Actions establish a relationship between snapshots of objects in the finishers and thus, transitively, devices define a relation between all snapshots of the products being produced. Parameters in actions represent a dimension of configuration that is of interest to operators of the devices. Constraints restrict instances of types and restrict the behavior of actions and devices, reducing the configuration space. We will now introduce the language concepts in more detail based on a specification for an example perfect binder such as described in Section 2.2.

Defined *types* are records of properties that model objects at locations in a device. In Figure 2.4, we define several types for the example perfect binder. Dimensions (widths, heights, lengths, distances) are modeled with integers with a precision of 0.1mm, such that an integer value of 10 stands for a length of 1mm. Types contain *defining properties* that are of a primitive type (boolean or integer) or of a defined type such that types can be nested. The nesting of types may not contain a cycle. Types optionally contain *constraints* and *derived properties*. Constraints restrict the inhabitants of a type. In Figure 2.4, the constraints (between square brackets), e.g., restrict sheets to have positive non-zero width and height. Derived properties are shorthands for expressions over other properties. Defining properties are required to instantiate a type. Derived properties are not required to instantiate a type and their values can

```

1  action ToMill(in: Stack, out: Stack) {
2    parameter millingDepth: int
3    [millingDepth ≥ 0] [out.width == in.width - millingDepth]
4    [out.height == in.height] [out.thickness == in.thickness]
5  }
6  action ToCrease(in: Sheet, out: CreasedSheet) {
7    [out.sheet.width == in.width] [out.sheet.height == in.height]
8    [out.spineFront.dir == 0] [out.spineBack.dir == 0]
9    [out.courtesyFront.dir == 1] [out.courtesyBack.dir == 1]
10   // Ensure a minimum distance between creases
11   parameter minDist: int [minDist > 0] [out.courtesyFront.off ≥ minDist]
12   [out.spineFront.off ≥ out.courtesyFront.off + minDist]
13   [out.spineBack.off ≥ out.spineFront.off + minDist]
14   [out.courtesyBack.off ≥ out.spineBack.off + minDist]
15   [out.courtesyBack.off ≤ in.width - minDist]
16 }
17 action ToCover(cover: CreasedSheet, book: Stack, out: PerfectBoundBook)
18 {
19   parameter d: int [0 ≤ d and d ≤ cover.sheet.height - book.height]
20   [out.book.width == book.width] [out.book.height == book.height]
21   [out.book.thickness == book.thickness]
22   [out.frontCover.width * 2 == cover.sheet.width - book.thickness]
23   [out.frontCover.height == cover.sheet.height]
24   [out.backCover.width * 2 == cover.sheet.width - book.thickness]
25   [out.backCover.height == cover.sheet.height]
26   [cover.spineFront.off * 2 == cover.sheet.width - book.thickness]
27   [cover.spineBack.off * 2 == cover.sheet.width + book.thickness]
28   parameter courtesyCreaseDist: int
29   [cover.courtesyFront.off * 2 ==
30     cover.sheet.width - book.thickness - courtesyCreaseDist * 2]
31   [cover.courtesyBack.off * 2 ==
32     cover.sheet.width + book.thickness + courtesyCreaseDist * 2]
33 }

```

Figure 2.5: The specification of actions for the example perfect binder in CSX. See Figure 2.3 for the dimensions used in this specification.

```

1 device ExamplePerfectBinder {
2   location bookIn : Stack location coverIn : Sheet
3   [1000 ≤ bookIn.height and bookIn.height ≤ 3000]
4   [2000 ≤ bookIn.width and bookIn.width ≤ 5000]
5   component toMill = ToMill(bookIn, milledBook) {
6     [millingDepth ≤ 30] // Max 3mm of milling
7     [bookIn.thickness < 170] // Max 17mm book thickness
8   }
9   location milledBook : Stack
10  component toCrease = ToCrease(coverIn, creasedCover) {
11    [minDist ≥ 50] // At least 5 mm between creases
12  }
13  location creasedCover : CreasedSheet
14  component toCover = ToCover(creasedCover, milledBook, out) {}
15  location out : PerfectBoundBook
16 }

```

Figure 2.6: The specification of the example perfect binder device in CSX.

be derived from other properties. A derived property expression may refer to the type’s properties and to other derived properties, but derived properties may not contain cyclic references. In Figure 2.4, *Stack* has a derived property *volume* which is defined in terms of defining properties.

Actions define a relation between locations. In Figure 2.5, we define several actions for the example perfect binder. The body of an action definition contains parameters and constraints that indicate the relations between its parameters.

Devices are sequences of *components* connected through *locations*. Components instantiate actions and can restrict or specify behavior further by adding constraints. Thus, action behavior is defined separately from specific instantiations in components. Therefore, actions are generic and potentially reusable between different device specifications. Limitations of a particular instance of an action in a device can be specified by adding constraints to the component. In Figure 2.6 we define a perfect binder device by instantiating several actions in components and by connecting them through the locations.

2.3.1 Configurations and Jobs

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

A job is an expression of intent for which a configuration needs to be found. Whereas configurations are a complete specification of locations and parameters, we could see jobs as a partial configuration. For example, a job could define the input and the output of the finisher. The remaining parts of

the configuration, i.e., the finishing parameters, need to be derived in order to instruct the finisher to realize the intent of the job. Different usage scenarios of a device lead to different jobs and approaches to configuration.

2.3.2 Exploration and Validation

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

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

```
1 scenario device ExamplePerfectBinder
2   config bookIn = Stack(2125,2970,50)
3   config out = PerfectBoundBook(Stack(2100,2970,50), Sheet(2100,2970),
4     Sheet(2100,2970)) {
5     [coverIn.width == 2100 + 2100 + 50]
6     [coverIn.height == 2970]
7     [toMill.millingDepth == 25]
8 }
```

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

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

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

2.4 DENOTATIONAL SEMANTICS

Because of the declarative characteristic of CSX, a translation to SMT constraints is natural. In this section, we define the denotational semantics of CSX that expresses a translation of CSX specifications to SMT constraints. Figure 2.7 contains the denotational semantics of CSX with the denotation expressed in MiniZinc [40, 41] definitions. Because we use MiniZinc in the implementation of CSX (Section 2.5), we also use it as the syntax for the denotation. The MiniZinc grammar can be found online¹.

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

$\llbracket S' \rrbracket_{S,N,R} = M$	Specification part S' of S translates to M in namespace N with location renaming R
$N = [x_1, x_2, \dots, x_n]$	Namespace N consisting of parts x_1 to x_n
$R = \{\dots, L_i \rightarrow L'_i, \dots\}$	Renaming of location names L_i to L'_i
$name([x_1, x_2, \dots, x_n]) = x_1_x2_ \dots _x_n$	Identifier for namespace $[x_1, x_2, \dots, x_n]$
Locations L , components C , constraints E , defining properties P , types T , action parameters PM .	
Devices	$\llbracket \text{device } d \{ L_1 \dots L_n, C_1 \dots C_m, E_1 \dots E_q, \dots \} \rrbracket_{S, \emptyset} =$ $\bigcup_{i=1}^n \llbracket [L_i] \rrbracket_{S, \emptyset} \cup \bigcup_{i=1}^m \llbracket [C_i] \rrbracket_{S, \emptyset} \cup \bigcup_{i=1}^q \llbracket [E_i] \rrbracket_{S, \emptyset} \quad (\text{DEVICE})$
Locations	$\frac{\text{type } T \{ P_1:T_1 \dots P_n:T_n, E_1 \dots E_m, \dots \} \in S}{\llbracket \text{location } L : T \rrbracket_{S, \emptyset} = \bigcup_{i=1}^n \llbracket [P_i:T_i] \rrbracket_{S, [L], \emptyset} \cup \bigcup_{i=1}^m \llbracket [E_m] \rrbracket_{S, [L], \emptyset}} \quad (\text{LOCATION})$ $\frac{T \in \{\text{int}, \text{bool}\}}{\llbracket [P:T] \rrbracket_{S,N,\emptyset} = \text{var } T : name(N ++ [P]) ;} \quad (\text{DEFPROP-PRIMTYPE})$ $\frac{\text{type } T \{ P_1:T_1 \dots P_n:T_n, E_1 \dots E_m, \dots \} \in S}{\llbracket [P:T] \rrbracket_{S,N,\emptyset} = \bigcup_{i=1}^n \llbracket [P_n:T_n] \rrbracket_{S,N++[P],\emptyset} \cup \bigcup_{i=1}^m \llbracket [E_m] \rrbracket_{S,N++[P],\emptyset}} \quad (\text{DEFPROP-DEFTYPE})$
Components	$\frac{\text{action } A(L_1:T_1^L \dots L_n:T_n^L) \{ \text{parameter } PM_1:T_1^P \dots \text{parameter } PM_m:T_m^P, E_1^A \dots E_q^A, \dots \} \in S}{\llbracket \text{component } C = A (L'_1 \dots L'_r) \{ E_1^C \dots E_s^C \} \rrbracket_{S, \emptyset} =} \quad (\text{COMP})$ $\bigcup_{i=1}^m \llbracket [\text{parameter } PM_m : T_m^P] \rrbracket_{S, [C], \emptyset} \cup \bigcup_{i=1}^q \llbracket [E_i^A] \rrbracket_{S, [C], R} \cup \bigcup_{i=1}^s \llbracket [E_s^C] \rrbracket_{S, [C], \emptyset}$ $\frac{T \in \{\text{int}, \text{bool}\}}{\llbracket [\text{parameter } PM:T] \rrbracket_{S,N,\emptyset} = \text{var } T : name(N ++ [PM]) ;} \quad (\text{PARAM})$
Constraints & References	$\llbracket [e] \rrbracket_{S,N,R} = \text{constraint } \llbracket [e] \rrbracket_{S,N,R}; \quad (\text{CONSTRAINT})$ $\frac{x \text{ is a defining property or parameter}}{\llbracket [x] \rrbracket_{S,N,R} = name(N ++ [x])} \quad (\text{DEFPROP-REF/PARAM-REF})$ $\frac{x \text{ is a location } \quad x \rightarrow x' \notin R}{\llbracket [x] \rrbracket_{S,N,R} = name(N ++ [x])} \quad (\text{LOCATION-REF})$ $\frac{x \text{ is a location } \quad x \rightarrow x' \in R}{\llbracket [x] \rrbracket_{S,N,R} = name(N ++ [x'])} \quad (\text{ACTIONLOCATION-REF})$ $\frac{x \text{ is a derived property with body } e}{\llbracket [x] \rrbracket_{S,N,R} = \llbracket [e] \rrbracket_{S,N,R}} \quad (\text{DERPROP-REF})$ $\llbracket [e.x] \rrbracket_{S,N,R} = \llbracket [e] \rrbracket_{S,N,R} + _x \quad (\text{PROJ})$

Figure 2.7: Denotational semantics of CSX, expressed in MiniZinc. We have omitted the rules for literals and arithmetic for brevity; they map one-to-one. ++ is namespace concatenation. + is identifier concatenation.

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

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

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

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

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

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

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

... in the specification.

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

2.5 IMPLEMENTATION

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

We have implemented the CSX language using Spoofox [20], a language workbench [8] that provides infrastructure for designing, implementing, and deploying DSLs by means of declarative specification of language aspects using meta-DSLs. We define the syntax of CSX in SDF₃ [42], a meta-language for multi-purpose syntax definition. From the CSX syntax definition, SDF₃ automatically derives a parser, pretty printer, syntax highlighting, and syntactic code completion. The parser yields abstract syntax trees (ASTs) on which we first apply desugaring. Desugaring, e.g., involves propagating the properties of a scenario to the tests within that scenario. The desugared ASTs are input to the static analysis and further transformations. We specify desugaring and other transformations using the Stratego [43] meta-language. Based on the language specification, Spoofox automatically generates an IDE for the language.

We define the CSX static semantics in NaBL₂ [44, 45]. NaBL₂ is a meta-language for specifying static semantics for languages from which name binding and type checking are automatically derived. Static semantic violations are reported interactively in the IDE. For CSX, this could be invalid composition of components in a device or incorrect type checking of constraint expressions. Interactive reporting of errors assists users of the language during specification writing.

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

To realize configuration space exploration, we implement a translation of CSX specifications to SMT constraints for which we can use existing solving techniques. In particular, we translate CSX to the MiniZinc constraint modeling language [40, 41]. MiniZinc is solver-independent, which enables us to use multiple solvers as a backend for CSX. In particular, we use solvers with the

theories of linear arithmetic and optimization modulo theories.

We implement the translation from CSX to MiniZinc as an AST-to-AST transformation using Stratego. In addition to the syntax definition of CSX, we have also defined the syntax of MiniZinc in Spoofax with SDF3². The syntax definitions of both languages generate an AST schema on which we define the Stratego transformation. After transforming a parsed CSX AST to a MiniZinc AST, the MiniZinc pretty printer generates concrete MiniZinc syntax from the AST.

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

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

The configuration space exploration pipeline serves two purposes in the IDE: test evaluation and inhabitation checking. For test evaluation, the configuration space of the device that is selected in the scenario is translated to MiniZinc and passed as an input to the pipeline. Additional constraints are added to reduce the configuration space, e.g., to configure the input or output location values, or parameters as specified in the scenario. If the scenario contains an objective, the objective is also mapped to MiniZinc and provided as input to the pipeline. The configuration that is returned by the pipeline is used to evaluate test expectations. This evaluation is done by a basic interpreter that evaluates expressions that should result into true. The expressions can contain references to parameters and location properties, and based on the name binding information the references are mapped to the corresponding value from the configuration. For failed test expectations we report an error which is marked with red underlining on the original specification using origin tracking [46].

The evaluation of tests and reporting of results is triggered in the IDE on file changes, resulting into an interactive experience. Additionally, the experience is improved by providing information while hovering over references to locations,

²<https://github.com/metaborgcube/metaborg-minizinc>

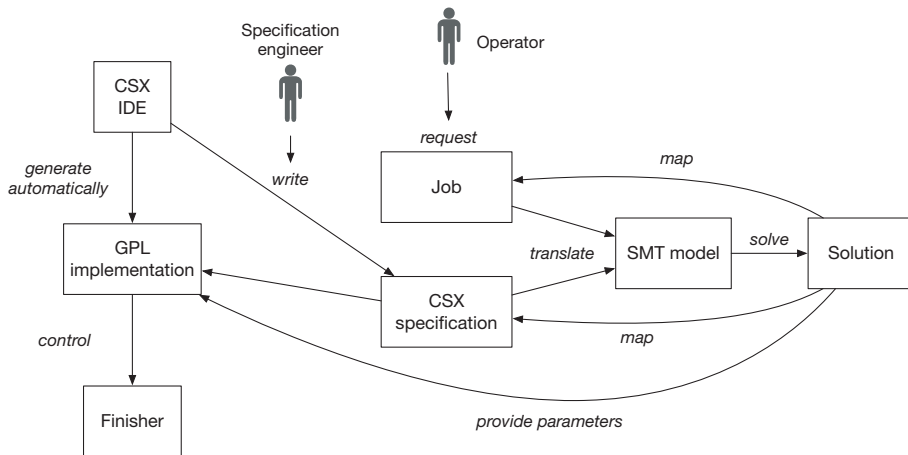


Figure 2.8: An architecture for applying CSX in control software. GPL stands for general-purpose programming language, such as C# or Java.

properties, and parameters in test expectations. The same interpretation approach as for test expectations is used to evaluate the expression being hovered over and the value is presented in a popup, giving the user insight into the configuration that is found.

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

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

While we have described the realization of a programming environment for CSX specifications, the eventual goal of CSX is to deploy control software to finishers. Figure 2.8 gives an overview of how configuration space exploration with CSX would fit in a realistic setting. The configuration space exploration component would be integrated with a software component, implemented using a general-purpose language, that provides a UI and that instructs low-level embedded software components.

2.6 EVALUATION

We evaluate CSX by modeling two realistic cases, a perfect binder and a booklet maker, and by benchmarking the configuration space exploration for a scenario with and without optimization. The perfect binder case corresponds to the example of Section 2.3. In the scenario without optimization, CSX derives the required input cover given an input book block and a desired output. In the scenario with optimization, CSX finds a configuration for the smallest size book the finisher can produce. The bookletmaker case concerns a finisher that performs rotating, stitching, folding, and trimming in order to produce a booklet from a stack of sheets. In the scenario without optimization, CSX finds the action parameters given an input and output. In the scenario with optimization, CSX finds a configuration that minimizes paper waste given only the desired output. Both specifications are based on realistic cases present at Canon Production Printing.

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

We set up a benchmark which makes use of Spoofox core, i.e., the core of Spoofox which enables integration of language components with Java, such that we can only execute the relevant part of the pipeline in the benchmark. For benchmarking, we use the JMH framework³. We executed the benchmarks on a server with two 32-core processors with a base frequency of 2.3GHz and 256GB RAM, running Ubuntu 20.04, using OpenJDK version 1.8.0_275-b01. From experimentation it appeared that the ORTools solver⁴ had best performance, and therefore we use this solver in the benchmarks. We use MiniZinc version 2.5.5 and ORTools version 9.0. We measure 10 iterations and average the result. In the benchmarks, we separately measure the translation time and solving time. We leave out parsing, name binding and type checking time, as they are minimal compared to translation and solving time.

Figure 2.9 shows the benchmarking results. For each scenario, solving time is in the order of 100's milliseconds. We consider sub-second performance as practical and therefore conclude that CSX's performance for the two cases we consider has practical performance for finding (optimal) configurations.

For specifying these devices in CSX, we have chosen a model of objects (sheets, stacks) with a certain level of detail. The bookletmaker and perfect binding cases translated in the SMT level into 32 and 29 variables and 56 and 58 constraints, respectively. Although we achieve useful configuration space

³<https://openjdk.java.net/projects/code-tools/jmh/>

⁴<https://developers.google.com/optimization>



Figure 2.9: The benchmarking results on a perfect binder and a booklet maker for a scenario of finding a configuration and for finding an optimal configuration.

exploration for these scenarios, it could be that in practice more detail has to be added to the model, which could also influence solving performance. By deploying CSX at Canon Production Printing, we aim to further evaluate whether CSX is adequate in modeling and integrating the full product line of finishers available and evaluate its usability for domain experts.

2.7 RELATED WORK

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

Keshishzadeh et al. use SMT solving for the validation of domain-specific properties to achieve fault detection early in the software development cycle. In particular, they develop a DSL with industrial application in a case on collision prevention for medical imaging equipment [47]. The approach includes delta debugging, i.e., an approach to trace causes of property violations and report them back to the specification in a systematic way. The work is related to CSX because it also uses SMT solving in the backend of a domain-specific language.

Voelter et al. use SMT solving with the Z3 solver for advanced error checking and verification in the KernelF language [48], a reusable functional language for the development of DSLs. Voelter et al. apply SMT solving successfully in a DSL on a case study for the domain of payroll calculations [49], i.e.,

for statically checking completeness and overlap of domain-specific switch-like expressions. Similarly to CSX, in this work, SMT solving is used in the backend of a domain-specific language for realizing static analyses. While the application of SMT was successful in the domain-specific case, the authors report difficulties in applying SMT solving generically in KernelF. The authors plan to develop a successor to KernelF that is realized with SMT solving completely.

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

Relational model finders are related to CSX in the sense that they map high-level specifications to constraints and map solutions back to the specification level. Alloy [50] is a specification language that applies finite model finding to check formal specifications of software. Alloy is backed by KodKod [51], a relational model finder for problems expressed using first-order logic, relational algebra, and transitive closures. In contrast to CSX, KodKod does not offer support for reasoning over data nor for optimization objectives. In CSX, the nature of specifications is not relational: manufacturing paths are fixed and we consider snapshots of the product being manufactured at different steps in the process.

AlleAlle [52] adds support for first-class data attributes and optimization to relational model finding. Similar to KodKod, Stoel et al. consider AlleAlle as an intermediate language. AlleAlle and CSX are related in the sense that both approaches take the data of problems into account and use SMT solving for model finding. While AlleAlle is an intermediate language generally targeting relational problems, CSX is a more domain-specific language in which relations are not a first-class concept. Similar to CSX, for AlleAlle it is unclear yet how to map reasons for unsatisfiability that are found in the constraint level back to the specification level.

Rosette [53] is a solver-aided programming language that supports verification, debugging, and synthesis. Rosette extends the Racket language with support for symbolic values that stand for, e.g., an arbitrary integer value. Such values translate to a constraint variable in the runtime. Rosette realizes verification and synthesis in the runtime by integrating its symbolic virtual machine with SMT solvers. Whereas in Rosette selected variables are replaced by symbolic values, in CSX all variables in the specification translate to constraint variables. Rosette is a general language tailored to program verification and synthesis whereas CSX is focused on a particular domain, i.e., manufacturing

systems, although we have only experimented with CSX in the digital printing domain.

Muli [54] is a constraint-logic object-oriented language that integrates constraint solving with object-oriented programming in the Java programming language. Muli extends Java's syntax with the `free` keyword for indicating symbolic values that translate to constraint variables in the runtime. Fragments of programs that are considered as search regions are executed non-deterministically, searching for concrete values for the constraint variables. The Muli runtime is based on a symbolic Java virtual machine that integrates constraint solvers. Muli only supports primitive types as constraint variables. Support for arrays and objects as constraint variables is listed as future work. CSX does support search on non-primitive types such as user-defined record types. Similar to how support for arrays is desired for Muli, support for lists is desired for CSX, but that is future work. Muli differs from CSX in the sense that Muli preserves the Java syntax and, by doing so, serves as a general-purpose programming language, whereas CSX introduces a new domain-specific language. In contrast to Muli, CSX supports optimization.

2.8 CONCLUSIONS

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

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

To improve the usability of the environments for configuration space exploration for operators, it would be useful to characterize the reduced configuration spaces for given jobs. In particular, when multi-objective optimization is relevant for objectives such as maximizing throughput and minimizing waste, it would be useful if CSX could indicate the tradeoff between these objectives.

2.9 APPENDIX: DECLARATIVE SEMANTICS

Specifications in CSX describe configuration spaces of devices. For a device specified in CSX, a configuration assigns values to the locations and parameters of the device. A valid configuration is a configuration that satisfies all constraints of the device. We describe the satisfiability relation of CSX by defining the declarative semantics of CSX in Figure 2.10. The rules follow the same pattern as the rules of the denotational semantics in Figure 2.7. The configuration space of a device corresponds to the set of all valid configurations that satisfy the declarative semantics.

$M \models_{S,R} S'$ $M \models_{S,R} S' \Rightarrow v$	<p>M models specification part S' of S with location renaming R</p> <p>M models specification part S' of S and evaluates to value v with location renaming R</p> <p>$R = \{\dots, L_i \rightarrow L'_i, \dots\}$ Renaming of location names L_i to L'_i</p> <p>$D(T)$ Domain of type T. $D(\text{bool}) = \{\top, \perp\}$, $D(\text{int}) = \mathbb{Z}$.</p> <p>Locations L, components C, constraints E, defining properties P, types T, action parameters PM.</p>
Devices	
$\frac{M \models_{S,\emptyset} L_1 \dots L_n \quad M \models_{S,\emptyset} C_1 \dots C_m \quad M \models_{S,\emptyset} E_1 \dots E_q}{M \models_{S,\emptyset} \text{device } d \{ L_1 \dots L_n, C_1 \dots C_m, E_1 \dots E_q, \dots \}} \quad (\text{DEVICE})$	
Locations	
$\frac{\text{type } T \{ P_1:T_1 \dots P_n:T_n, E_1 \dots E_m, \dots \} \in S \quad M.L = v \quad v \models_{S,\emptyset} P_1:T_1 \dots P_n:T_n \quad v \models_{S,\emptyset} E_1 \dots E_m}{M \models_{S,\emptyset} \text{location } L : T} \quad (\text{LOCATION})$	
$\frac{T \in \{\text{int}, \text{bool}\} \quad M.P \in D(T)}{M \models_{S,\emptyset} P : T} \quad (\text{DEFPROP-PRIMTYPE})$	
$\frac{\text{type } T \{ P_1:T_1 \dots P_n:T_n, E_1 \dots E_m, \dots \} \in S \quad M.P = v \quad v \models_{S,\emptyset} P_1:T_1 \dots P_n:T_n \quad v \models_{S,\emptyset} E_1 \dots E_m}{M \models_{S,\emptyset} P : T} \quad (\text{DEFPROP-DEFTYPE})$	

Figure 2.10: Declarative semantics of CSX (continued on next page). We have omitted the rules for literals and arithmetic for brevity; they map one-to-one. We define models M recursively as $(M, x = v)$ in which value v binds to x and with the empty model \emptyset as base case. We define model projection as $(M, x = v).x = v$ and $(M, x = v).y = M.y$ if $x \neq y$. $e \Rightarrow v$ indicates that syntactic expression e evaluates to value v . Values are booleans (\top and \perp), integers, and models.

Components	
$\frac{\begin{array}{l} \text{action } A(L_1:T_1^L \dots L_n:T_n^L) \\ \{\text{parameter } PM_1:T_1^P \dots \text{parameter } PM_m:T_m^P, E_1^A \dots E_q^A, \dots\} \in S \\ R = \{L_1 \rightarrow L'_1, \dots, L_n \rightarrow L'_n\} \quad M.C = v \\ v \models_{S,\emptyset} \text{parameter } PM_1:T_1^P \dots \text{parameter } PM_m:T_m^P \\ v \models_{S,R} E_1^A \dots E_q^A \quad v \models_{S,\emptyset} E_1^C \dots E_s^C \end{array}}{M \models_{S,\emptyset} \text{component } C = A (L'_1 \dots L'_n) \{ E_1^C \dots E_s^C \}} \quad (\text{COMP})$	
$\frac{T \in \{\text{int}, \text{bool}\} \quad M.PM \in D(T)}{M \models_{S,\emptyset} \text{parameter } PM : T} \quad (\text{PARAM})$	
Constraints & References	
$\frac{M \models_{S,R} e \Rightarrow v \quad v = \top}{M \models_{S,R} [e]} \quad (\text{CONSTRAINT})$	
$\frac{x \text{ is a defining property or parameter} \quad M.x = v}{M \models_{S,R} x \Rightarrow v} \quad (\text{DEFPROP-REF/PARAM-REF})$	
$\frac{x \text{ is a location} \quad x \rightarrow x' \notin R \quad M.x = v}{M \models_{S,R} x \Rightarrow v} \quad (\text{LOCATION-REF})$	
$\frac{x \text{ is a location} \quad x \rightarrow x' \in R \quad M.x' = v}{M \models_{S,R} x \Rightarrow v} \quad (\text{ACTIONLOCATION-REF})$	
$\frac{x \text{ is a derived property with body } e \quad M \models_{S,R} e \Rightarrow v}{M \models_{S,R} x \Rightarrow v} \quad (\text{DERPROP-REF})$	
$\frac{M \models_{S,R} e \Rightarrow v1 \quad v1.x = v2}{M \models_{S,R} e.x \Rightarrow v2} \quad (\text{PROJ})$	

Figure 2.10: Declarative semantics of CSX (continued).

2.10 APPENDIX: INHABITANCE

The CSX syntax allows to define types, actions, and devices without inhabitants. For example, the following type is not inhabited:

```
1 type T { i: int [i != i] }
```

Since there are no valid configurations for such definitions, we want to detect and report uninhabited definitions. Specifications with definitions that are not inhabited, i.e., there are no models for their instantiations, are not useful in practice. Therefore, we restrict CSX such that types, actions, and devices must be inhabited. Below, we define inhabitation in terms of the satisfiability relation of Figure 2.10.

A type T is inhabited if there exists a model M that is a value for a arbitrary location L of type T and that satisfies the specification of the type:

$$\frac{\text{arbitrary name } L}{\exists M \models_{\emptyset, \emptyset} \text{location } L : T}$$

An action A is inhabited if there exists a model M that satisfies an instance of the action, i.e., a valuation its for parameters and that satisfies the specification of the action:

$$\frac{\begin{array}{l} M \models_{\emptyset, \emptyset} \text{location } L_1 : T_1^L \dots \text{location } T_n : T_n^L \\ R = \{L_1 \rightarrow L_1, \dots, L_n \rightarrow L_n\} \\ M \models_{\emptyset, \emptyset} \text{parameter } PM_1 : T_1^P \dots \text{parameter } PM_m : T_m^P \\ M \models_{\emptyset, R} E_1^A \dots E_q^A \end{array}}{\exists M \models_{\emptyset, \emptyset} \text{action } A(L_1 : T_1^L \dots L_n : T_n^L) \\ \{\text{parameter } PM_1 : T_1^P \dots \text{parameter } PM_m : T_m^P, E_1^A \dots E_q^A, \dots\}}$$

A device d is inhabited if there exists a model M that satisfies the device: $\exists M \models_{S, \emptyset} \text{device } d \{ \dots \}$.

Inhabitation corresponds to satisfiability in the SMT domain. Unsatisfiability of an SMT model for a type, action, or device indicates the definition is not inhabited. For a device, it means the configuration space is empty. The denotational semantics in Figure 2.7 defines a translation from the perspective of a device. We build on this translation to define rules for checking inhabitation of types and actions in Figure 2.11.

For inhabitation checking of types and actions, we can reuse the rules but have to provide an artificial context for the translation. For example, for inhabitation checking of type T , we check satisfiability of the SMT model for an instance of the type in a arbitrary location (LOCATION-INHAB). The type is inhabited if the SMT model for an instance of the type in an arbitrary location is satisfiable. For inhabitation checking of actions, we take a similar approach (ACTION-INHAB). Instead of taking a single arbitrary location, we instantiate locations for all location parameters, and use them to instantiate the action A for an arbitrary component C .

$$\begin{array}{c}
\frac{\text{arbitrary name } L}{\llbracket \text{location } L : T \rrbracket_{S, [], \emptyset}} \quad (\text{LOCATION-INHAB}) \\
\\
\begin{array}{c}
\text{action } A(L_1 : T_1^L \dots L_n : T_n^L) \{ \\
\text{parameter } PM_1 : T_1^P \dots \text{parameter } PM_m : T_m^P, \\
E_1^A \dots E_q^A, \dots \} \in S \quad R = \{L_1 \rightarrow L_1, \dots, L_n \rightarrow L_n\} \\
\text{arbitrary name } C
\end{array} \\
\hline
\begin{array}{c}
\bigcup_{i=1}^n \llbracket \text{location } L_i : T_i^L \rrbracket_{S, [], \emptyset} \cup \\
\bigcup_{i=1}^m \llbracket \text{parameter } PM_m : T_m^P \rrbracket_{S, [C], \emptyset} \cup \\
\bigcup_{i=1}^q \llbracket E_i^A \rrbracket_{S, [C], R}
\end{array} \quad (\text{ACTION-INHAB})
\end{array}$$

Figure 2.11: Denotational semantics for inhabitation checking, building on the rules of Figure 2.7.

CSX 2.0: Taming Complexity of Industrial Printing Systems Using a Constraint-Based DSL — An Industrial Experience Report

3

ABSTRACT

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

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

Based on: Jasper Denkers, Marvin Brunner, Louis van Gool, Jurgen J. Vinju, Andy Zaidman, and Eelco Visser. "Taming complexity of industrial printing systems using a constraint-based DSL: An industrial experience report". In: *Software: Practice and Experience* (2023). DOI: 10.1002/spe.3239.

3.1 INTRODUCTION

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

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

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

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

However, modeling a digital printing system — including all details of the mechanics — in a generic constraint modeling language is tedious, because it involves low-level modeling. Using a domain-specific language (DSL) for modeling configuration spaces has the potential to tackle this issue. With a DSL, we can automate the transformation of more high-level models of printing systems to constraint models. On these generated constraint models, we use constraint solvers to find (optimal) configurations and realize automatic configuration space exploration. The modeling of printing systems in the DSL is in terms of the printing domain and abstracts over low-level and repetitive constraint modeling, making the modeling task feasible in practice.

```

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

```

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

```

1 var 0..10 : a_sheets_size;
2 array [1..10] of var int : a_sheets_width;
3 constraint forall(i in 1..10) (i > a_sheets_size → a_sheets_width[i] =
4     0);
5 array [1..10] of var int : a_sheets_height;
6 constraint forall(i in 1..10) (i > a_sheets_size → a_sheets_height[i]
7     = 0)

```

(b) MiniZinc [40] constraint model for device D with variables for the size and properties of the stack's sheet list, for an upper bound of 10 sheets. The constraints on lines 3–4 frame variables that are not considered in the sheet list to a default value (0 for integers).

```

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

```

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

```

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

```

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

Figure 3.1: An artificial CSX model (a), its translation to constraints (b), a solution for the constraint model (c), and the solution mapped back to a configuration on the CSX level (d).

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

In the previous chapter, we have developed CSX (Configuration Space eXploration), a DSL for modeling digital printing systems, automatically generating constraint models from device models, mapping solutions back to the domain of printing configurations, and deriving environments for configuration space exploration. Figure 3.1 depicts the translation of CSX to constraints and the mapping of a solution to a device configuration for an example model. Our hypothesis is that CSX is an effective and scalable method for creating control software for digital printing systems. With sufficient coverage and practical solving performance, it has the potential to improve the current state of control software development for printing systems by adding functionality (introducing configuration space exploration that is automatic and complete) and reducing software engineering complexity.

In addition to validating the concepts of CSX, our objective is to evaluate CSX's practical applicability. This has guided our approach in two ways. First, we design the language from the perspective of the user, top-down, meaning that we do not restrict language features before having substantiation for such restrictions from practice. Second, we use MiniZinc [40] as a facade for different underlying SMT solvers [4], as we did not want an early design decision for a specific solver to later hinder our experimentation opportunities.

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

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

- CSX 2.0, which adds language support for generic lists, geometrical con-

structs, and functional-style operators.

- An evaluation of CSX 2.0 in an industrial context.
- Lessons learned on using a constraint-based DSL in an industrial context.

3.2 INDUSTRIAL PRINTING AND FINISHING SYSTEMS

3.2.1 *Printing and Finishing*

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

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

As an example, we take an edge stitching device such as depicted in Figure 3.2. This device takes a stack of sheets of limited sizes, stitches the stack at the right edge, and optionally rotates the stitched stack before outputting it. Table 3.1 depicts four scenarios of configuration space exploration for this device.

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

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

Scenario D is not possible. It requires A3 sheets in landscape orientation with the stitch at the top edge. The stitch at the top edge requires to rotate 90

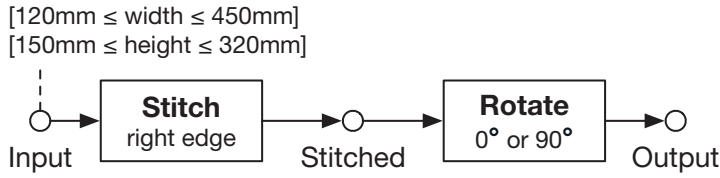


Figure 3.2: Schematic overview of the finishing steps of a simplified edge stitching device. Dots indicate locations at which we consider a snapshot of the stack of sheets that is stitched. The input is limited to the sizes of sheets it can handle. Rectangles represent the actions that are performed on the objects. The device can only perform a right-edge stitch, and can subsequently rotate the stack by 0 or 90 degrees.

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

Table 3.1: Scenarios of configuration space exploration for the edge stitching device from Figure 3.2. Each row (A-D) represents a scenario. The middle four columns indicate the values for a configuration corresponding to the scenario. Question marks (?) indicate unknown values, for information that needs to be derived by configuration space exploration. Width (w), height (h), and stitch edge (e) are abbreviated and the millimeter unit is omitted.

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

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

In principle, we can use a general-purpose programming language to model printing systems, and to implement algorithms for finding configurations

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

A natural starting point for modeling a printing system, or a manufacturing system in general, is to identify locations through which objects of the manufacturing process pass. Next, we can define each action with its parameters that alter the manufacturing objects from one intermediate location into the next. By doing so, each location represents a snapshot of the process that transforms the input step-wise. These actions and locations correspond to those in Figure 3.2.

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

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

We need an *analytical* approach — different than the constructive approach based on simulation and manual heuristics — that automatically derives the complete configuration space of a device, given only a partial description of a configuration. Because the configuration space is large and actions are interdependent, this is hard to achieve with manual programming effort, especially given the large variety of printing systems. This is where we can leverage the power of constraint solving, which seems a natural fit for configuration space exploration. By specifying the configuration spaces in

terms of constraints, we can use constraint solvers to find configurations. It does not matter anymore for which scenario — forward, backward, or anything in between such as finding parameters given an input and output. The problem with modeling digital printing systems directly in a generic constraint modeling language, however, is that it is tedious and repetitive work.

3.2.2 Requirements

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

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

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

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

3.2.3 CSX: *Configuration Space eXploration*

In the previous chapter, we have developed CSX 1.0: a language and environment that serves as an interface to constraint programming specific to the printing domain, abstracting over the complexity of developing control software in two ways. First, CSX offers domain-specific constructs that abstract over low-level details. Such details do not need to be rethought each time a new device is modeled. In CSX, a library of actions can be built, which can be reused in device models. Second, by leveraging the power of constraint solvers to find configurations for printing devices, control software engineers do not have to manually develop algorithms to find configurations. Therefore, CSX promises to tackle two of the most challenging aspects of developing control software for printing systems.

In CSX, we model printing systems by modeling intermediate locations of the manufacturing process and configuration parametricity. We will now further introduce CSX's language concepts using Figure 3.3, an example CSX model of an edge stitching device.

User-defined record-types. In CSX, we use *user-defined record-types* to model the objects in the manufacturing process. This concerns the input, output, and snapshots of the products at intermediate locations. Instead of specifying all properties individually, CSX users can define record-like types such as sheets and stacks. In the example, `Sheet` and `StitchedStack` are user-defined types (lines 1–9). User-defined types are records of properties which can be either defining properties or derived properties. We can use this abstraction

```

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

```

Figure 3.3: CSX model of an edge stitching device (schematically depicted in Figure 3.2) that has a list of sheets as input, stitches them in the right edge, and optionally rotates the stitched stack 90 degrees before it leaves the machine as output. Integer dimensions in this model represent 1/10mm.

to incorporate objects such as sheets and stacks in a device model to model the snapshots of printing products. Besides user-defined types, the language supports integers and booleans as primitives.

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

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

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

Scenarios & Tests. In CSX models, we can also define *tests* for devices. Such tests are used to specify configuration space exploration scenarios with assertions to validate the models. Figure 3.4 lists several tests for the example model. By using *scenarios*, a restricted configuration space can be considered for multiple tests nested in the scenario. The assertions can simply expect there to be a configuration (`succeeds`), no configuration (`fails`), or expect something more specific using the constraint notation. For example, the first test expects a rotation of 0 degrees (line 14) and the second test expects a rotation by 90 degrees (line 19).

We have implemented the CSX language and IDE using the Spoofox language workbench [20]. We express the translations of CSX models to SMT models in MiniZinc [40]. MiniZinc is a generic and solver-independent constraint modeling language, which allows us to use various solvers for finding configurations. Both the translation of CSX models and tests to MiniZinc and the mapping of SMT solutions back to the CSX level are implemented using the Stratego transformation language [43]. Tests are evaluated interactively in the IDE, i.e., a test is re-evaluated automatically if and only if it or the device under test has changed. Figure 3.5 depicts how feedback on tests is presented in the IDE.

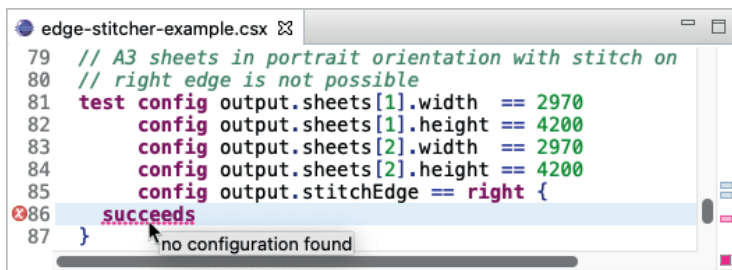
Figure 3.6 depicts an architecture that applies CSX to realize control software. We have implemented the CSX language and IDE and all components relevant

```

1 // All tests are for the EdgeStitcher device and with two sheets in the
  output
2 scenario device EdgeStitcher
3     config size(output.sheets) == 2 {
4
5 // A4 sheets in portrait orientation
6     scenario config output.sheets[1].width == 2100
7         config output.sheets[1].height == 2970
8         config output.sheets[2].width == 2100
9         config output.sheets[2].height == 2970 {
10
11 // Stitch on right edge requires no rotation
12     test config output.stitchEdge == right {
13         [rotation == rot0]
14     }
15
16 // Stitch at top edge requires rotation of 90 degrees
17     test config output.stitchEdge == top {
18         [rotation == rot90]
19     }
20
21 }
22
23 // A3 sheets in portrait orientation with stitch on
24 // right edge is not possible
25     test config output.sheets[1].width == 2970
26         config output.sheets[1].height == 4200
27         config output.sheets[2].width == 2970
28         config output.sheets[2].height == 4200
29         config output.stitchEdge == right {
30     fails
31 }
32
33 // A3 sheets in landscape orientation with stitch on
34 // right edge is possible
35     test config output.sheets[1].width == 4200
36         config output.sheets[1].height == 2970
37         config output.sheets[2].width == 4200
38         config output.sheets[2].height == 2970
39         config output.stitchEdge == right {
40     succeeds
41 }
42
43 }

```

Figure 3.4: Tests accompanying the CSX model of Figure 3.3.



```
edge-stitcher-example.csx
79 // A3 sheets in portrait orientation with stitch on
80 // right edge is not possible
81 test config output.sheets[1].width == 2970
82   config output.sheets[1].height == 4200
83   config output.sheets[2].width == 2970
84   config output.sheets[2].height == 4200
85   config output.stitchEdge == right {
86 succeeds
87 }
```

Figure 3.5: An example of interactive validation in CSX. The test (a modified version of the third test in Figure 3.4) incorrectly expects CSX to find a configuration. The CSX IDE reports that this expectation is incorrect using an error marker. While hovering over the incorrect expectation, the popup indicates that no configuration was found.

for automatic configuration space exploration. The deployment of CSX with code generation for communication with embedded software in devices and the integration with user interfaces is future work. For further details about the semantics and implementation of CSX 1.0, we refer to the previous chapter. The technical contribution of the current chapter focuses on extending the coverage of CSX within the existing framework, which we discuss next.

3.2.4 Coverage Gaps

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

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

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

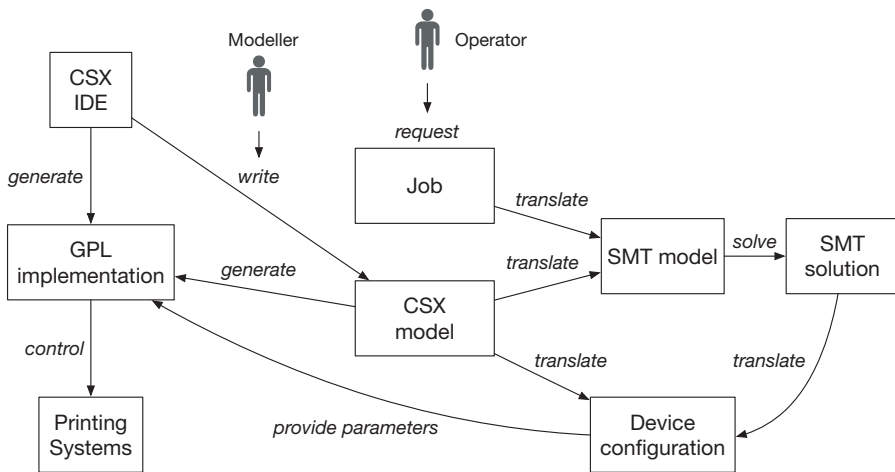


Figure 3.6: An architecture in which CSX is applied to realize control software for digital printing systems. GPL stands for a general-purpose programming language such as C#.

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

3.3 INCREASING DOMAIN COVERAGE

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

3.3.1 Non-Uniform Stacks of Sheets

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

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

<pre> 1 var xs: list<int> 2 var ys: list<int> 3 4 reverse xs into ys </pre>	<pre> 1 size(ys) == size(xs) 2 xs.forall { x => 3 ys[size(xs) + 1 - index] = x 4 } </pre>
---	--

(a) reverse

<pre> 1 var xs: list<int> 2 var ys: list<int> 3 4 map xs with x => x + 1 into ys </pre>	<pre> 1 size(ys) == size(xs) 2 xs.forall { x => 3 ys[index] = x + 1 4 } </pre>
--	---

(b) map

<pre> 1 var xs: list<int> 2 var ys: list<int> 3 var z: int 4 5 append z after xs into ys </pre>	<pre> 1 size(ys) == size(xs) + 1 2 xs.forall { x => 3 ys[index] = x 4 } 5 last(ys) == z </pre>
---	---

(c) append

<pre> 1 var xs: list<int> 2 var ys: list<int> 3 var z: int 4 5 prepend z before xs into ys </pre>	<pre> 1 size(ys) == size(xs) + 1 2 first(ys) == z 3 xs.forall { x => 4 ys[index + 1] = x 5 } </pre>
---	--

(d) prepend

Figure 3.7: Translation of CSX's list operations (left-hand sides) into forall (right-hand sides, also CSX). The target models (right-hand sides) have variable declarations omitted for brevity.

In CSX 2.0, we add support for the generic `list<T>` type for lists with elements of type `T`. Lists in CSX 2.0 do have a variable size and can be instantiated for both primitive and user-defined types. Figure 3.1a shows an example that models a stack as a list of sheets (line 2).

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

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

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

Figure 3.1 shows an example of a stack that is modeled as a list of sheets and how that translates to constraints, expressed in MiniZinc. The variable `a_sheets_size` represents the size of the list of sheets. The domain for this variable (`0..10`, i.e. an integer value in the range 0 to 10) represents the possible sizes of the list (given that the upper bound n_{max} is 10). Sheets have two properties: `width` and `height`, both of type `int`. The target model contains an array with variables for each property, denoted in MiniZinc with, e.g., array `[1..10]` of `var int : a_sheets_width` for the width property. Again, the size of this array is determined by the upper bound on the size of lists.

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

To make lists in CSX practical, we add operations on lists such as `reverse`, `append`, `prepend`, and `map`. Many of such expressions over lists can be expressed by translation into a `forall` construct. The `forall` and `exists` quantifiers are common constructs in (functional) languages that support lists. The `forall` operator is used to express whether a predicate holds for all elements in a collection. We implement the `forall` construct and use it as a core construct which other operations translate to, see Figure 3.7.

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

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

We call the operations in Figure 3.7 to be in *predicate-style*, i.e., the operators enforce a predicate over multiple (list) variables. For example, `reverse xs into ys` evaluates to true if `xs` is the reverse of `ys`. The type of the predicate-style `reverse`, `append`, `prepend`, and `map` operations is therefore boolean.

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

While a functional language would dynamically allocate memory for intermediate values of such chains of operations at runtime, in constraint programming the variables need to be known upfront. In Section 3.3.3 we describe an algorithm for introducing intermediate variables where needed. That will allow writing, e.g., `ys == reverse(xs)`, which will then first translate into the predicate-style variant (`reverse xs into ys`), which in turn will translate into a `forall` expression.

3.3.2 Geometrical Constructs

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

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

```

1 // Artificial device model that transforms an edge
2 device D {
3     location e1: edge
4     location e2: edge
5     location o: orientation
6
7     [e2 == orientate(e1, o)]
8 }
9 test device D config e1 == top config e2 == bottom {
10     [o == rot180 or o == flip0]
11 }

```

Figure 3.8: An artificial CSX model with an orientation parameter o that transforms edge $e1$ into $e2$.

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

To effectively use orientations, we introduce high-level constructs for concepts that can be orientated such as edges. Again, edges could already be modeled using primitives or user-defined types, but having dedicated constructs enables us to implement concise operations for them with orientations. An edge is one of `top`, `right`, `bottom`, or `left`, which are represented as two-dimensional vectors in the constraint domain. For example, `top` corresponds to $\begin{bmatrix} 0 & 1 \end{bmatrix}$.

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

Figure 3.8 depicts an artificial CSX model for a device that transforms an edge over an orientation. Figure 3.10 depicts the corresponding MiniZinc target model. The translation of geometrical constructs in CSX to MiniZinc makes use of a prelude, which is depicted in Figure 3.9. This prelude is MiniZinc code that represents data types, predicates, and functions over the geometrical constructs that are referenced in the target (MiniZinc) model of a specific CSX model.

An edge is represented with a two-dimensional vector in MiniZinc, using the type array `[1..2]` of `var -1..1`; an array with indices in the range `1..2`

```

1  % Prelude for target model (in MiniZinc)
2
3  % Constants for the eight relevant orientations
4  array[1..2,1..2] of var -1..1: Rot0    = [| 1, 0| 0, 1|];
5  array[1..2,1..2] of var -1..1: Rot90  = [| 0, 1|-1, 0|];
6  array[1..2,1..2] of var -1..1: Rot180 = [| -1, 0| 0, -1|];
7  array[1..2,1..2] of var -1..1: Rot270 = [| 0, -1| 1, 0|];
8  array[1..2,1..2] of var -1..1: Flip0   = [| 1, 0| 0, -1|];
9  array[1..2,1..2] of var -1..1: Flip90  = [| 0, -1|-1, 0|];
10 array[1..2,1..2] of var -1..1: Flip180 = [| -1, 0| 0, 1|];
11 array[1..2,1..2] of var -1..1: Flip270 = [| 0, 1| 1, 0|];
12 % Constants for the four relevant edges
13 array[1..2] of var -1..1: Top    = [| 0, 1|];
14 array[1..2] of var -1..1: Right = [| 1, 0|];
15 array[1..2] of var -1..1: Bottom = [| 0, -1|];
16 array[1..2] of var -1..1: Left  = [| -1, 0|];
17 % Predicate to restrict an orientation's matrix
18 predicate isOrientation(array[1..2,1..2] of var -1..1: o) =
19   o = Rot0 ∨ o = Rot90 ∨ o = Rot180 ∨ o = Rot270 ∨
20   o = Flip0 ∨ o = Flip90 ∨ o = Flip180 ∨ o = Flip270;
21 % Predicate to restrict an edges's matrix
22 predicate isEdge(array[1..2] of var -1..1: e) =
23   e = Top ∨ e = Right ∨ e = Bottom ∨ e = Left
24 % Function for orientating an edge
25 function array[1..2] of var -1..1: orientateEdge(
26   array[1..2] of var -1..1 : e,
27   array[1..2,1..2] of var -1..1 : o
28 ) =
29   array1d(1..2, [
30     e[1] * o[1,1] + e[2] * o[1,2],
31     e[1] * o[2,1] + e[2] * o[2,2]
32   ]);

```

Figure 3.9: The prelude MiniZinc code for geometrical constructs that is added to target models.

```

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

```

Figure 3.10: The device-specific target model corresponding to Figure 3.8, making use of the prelude (Figure 3.9).

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

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

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

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

3.3.3 *Functional-Style Operators*

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

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

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

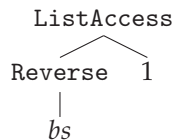
results.

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

Let's take the following CSX spec:

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

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



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

It will be translated into:

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

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

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

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

Finally, the transformation step for predicate-style operations on lists (as defined in Section 3.3.1) transforms the expression into a `forall` construct:

```

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

```

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

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

3.4 INDUSTRIAL EVALUATION

Having designed the CSX language and associated environment for configuration space exploration, we now evaluate the industrial application of CSX 2.0 at Canon Production Printing. We explore whether CSX as a constraint-based language is effective for modeling printing systems and realizing automatic configuration space exploration. We evaluate whether CSX meets our requirements (Section 3.2.2) on domain coverage, configuration accuracy, and configuration performance. Additionally, we evaluate the relevance of the approach. In particular, we consider the following evaluation questions:

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

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

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

Relevance Is it *sufficient* to use CSX to achieve automatic configuration space

exploration and is it *necessary* to use CSX instead of directly modeling printing systems in a generic constraint modeling language such as MiniZinc?

In the remainder of this section, we describe our evaluation method and discuss results per question. Although this chapter presents an extension of CSX 1.0 (Chapter 2), we evaluate CSX 2.0 as a whole — not only the new features.

3.4.1 Domain Coverage

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

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

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

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

During think-aloud co-design sessions [55] the two participants gather data for evaluating the language design. The participants model the edge stitching device by gradually including aspects of increasing complexity, selected by the domain expert. In particular, these sessions follow the following protocol:

- The participants communicate via a video call. The first participant has the

CSX IDE open and shares the screen with the other participant, such that both participants can see the IDE and models.

- The participants perform iterations of modeling in sessions of two hours.
- For each iteration, the domain expert selects an aspect of the device to model. Initially, the domain expert selects the most simple aspect of the device. The judgment of the domain expert is leading in gradually expanding the level of detail of the model. For new iterations, the domain expert expands an aspect with more detail or selects a new aspect. Each iteration starts in a new CSX file by copying the previous file, initially starting with an empty file.
- The participants engage in a think-aloud conversation [55] on which properties to consider and which design decisions to make during the process. The first participant writes the CSX code that corresponds to the consensus of the participants on how to model the selected aspect.
- The participants document the considerations and design decisions by taking notes in comments of the CSX code such that the considerations can be revisited when discussing the evaluation questions.
- The participants validate the model by writing tests, and revert to fixing the model if tests reveal flaws in the model.
- The participants repeat this process until the domain expert concludes that the device is modeled with a level of detail that is sufficient for realizing control software.

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

Results. The participants performed six co-design sessions of two hours. In some sessions, the participants worked on multiple iterations, and some iterations were based on work from multiple sessions. The sessions resulted in seven iterations of CSX models, of which the last contained the final model of the edge stitching device. Table 3.2 gives an overview of the aspects that were included in each iteration.

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

General Observations. Before starting on the first CSX model, the participants realized that they should determine the scope of what they would include in the model. The most high-level question in that regard is whether the

Table 3.2: The aspects that were included in each iteration of the co-design evaluation sessions. The first attempt at modeling input trays (iteration 3) was incorrect and was modeled again from scratch (iteration 6).

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

model should start before or after the printing device. Although CSX has been originally designed with the intention to model and integrate finishing equipment, there is also utility in including part of the printing device in CSX models (*GO 1*).

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

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

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

Uniform stacks of sheets. In order to start simple and gradually expand the level of detail in the model, the participants chose to start with modeling stacks of sheets that are uniform. This has a restrictive implication: all sheets in a stack are considered equal by design. Although this is an oversimplification

```

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

```

Figure 3.11: The CSX model resulting from iteration 1. A simple device that takes a uniform stack with a size (width and height) and number of sheets as input and outputs the same stack. Hardware limitations on the size of the input stack are captured in constraints. Two tests cover a succeeding and failing scenario.

which is not realistic, the participants considered it a good starting point.

Figure 3.11 (lines 1–5), from the first modeling iteration, depicts the type definition for a uniform stack of sheets. Many properties could be included in sheets, but the participants started with a simple representation of sheets with only a size (width and height). An additional property `sheets` indicates how many sheets are in the stack.

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

The integer type in CSX has a domain of both positive and negative integer values. Since the size of the sheets and the number of sheets cannot be negative, the participants added constraints to the model to restrict the instances (Figure 3.11, lines 2–4, between square brackets).

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

The participants observed that the equality between input and output (Figure 3.11, line 13) is in terms of stacks, not in terms of the individual properties of stacks. Equality can thus be defined on a level of abstraction that corresponds to the objects modeled in CSX. This is in contrast to a low-level constraint modeling language, in which one would need to define equality with low-level constraints that compare each property of the stack individually (*COVERAGE 3*).

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

The first iteration only contained an input and output location, both of type `Stack` (Figure 3.11). This is an oversimplification of the actual device: the model does include the physical limitations of the device, but it does not contain the different input trays, the stack cannot consist of different types of sheet (non-uniformity), and stitching and the possibility to orientate the sheets before and after stitching are not included.

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

The participants observed that the stack of sheets at the input location of the device could be interpreted in two ways: they represent the total number of sheets that are in the input trays, or they represent the sheets in the input trays that will be used in the configuration for a single product. Alternatively,

the configuration could also be used for multiple products in one job. In this model, the participants modeled the configuration space for single products. Therefore, the input of the device in the model represents the sheets of paper for a single job; there could be more paper in the physical tray.

The participants noticed that in the design process, they did not use actions (see Section 3.2.3) yet to factor out common pieces of behavior, but modeled everything directly in a device. CSX supports actions for building a library of printing behavior that can be shared between many device models, but they were not used in the study (GO 6).

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

Physical Limitations. The participants added physical limitations of the device in the first iteration (Figure 3.11, lines 10–11). The constraints in square brackets express the device’s physical limitations with respect to the minimum and maximum size of sheets that it can handle. In this iteration, no maximum on the number of sheets was modeled. Note that the constant values that indicate the minimum and maximum width and height are integer values that represent a dimension for the precision chosen in this model. For example, the constraint `input.width <= 4500` indicates that the maximum width is 450mm.

Validation. Having a first simple model of the device, the participants wrote two tests to validate the physical constraints (Figure 3.11, lines 18–31). The first test checks that for a given input that is accepted within the physical constraints, the output contains the same stack. The second test checks that for an input that is too small, no configuration can be found (indicated with fails).

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

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

The participants observed that the modeling of objects in tests is still at

```

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

```

Figure 3.12: The CSX model resulting from iteration 2 which includes the aspect of *non-uniform* stacks of sheets. The physical limitations of the device are expressed using a forall expression on the list of sheets in the input stack.

a low level of abstraction (*COVERAGE 4*). For example, to specify an input sheet object, one needs to specify each property of the sheet with individual constraints. Figure 3.18a depicts this: the test contains a config instance per property of the sheet that is relevant for the test. In this case, the thickness of the sheet is not relevant to the test and is thus omitted.

Non-uniform stacks of sheets. In iteration 2 (Figure 3.12), the participants aimed to increase the level of detail of the model by allowing stacks to be non-uniform. To do so, the participants refactored the model to use CSX’s list construct for stacks of sheets (line 7). This enables to model non-uniform stacks, i.e., the sheets in the stack can have different properties (*COVERAGE 5*). Since lists can have a variable size, the stacks can have a variable number of sheets. The participants noticed that a limitation of CSX is that although the upper bound is configurable, all lists get the same upper bound (*GO 10*).

Input trays. In iteration 3 (Figure 3.13), the participants first attempted to model input trays by combining the uniform stacks and non-uniform stacks. There are two input tray locations of type `UniformStack`. The case focuses on forming a non-uniform stack of sheets from the two input trays of uniform stacks of sheets. The idea behind the approach was as follows: the output

```

1 type Sheet {
2   width: int, [width > 0],
3   height: int, [height > 0]
4 }
5 type UniformStack {
6   width: int, [width > 0],
7   height: int, [height > 0],
8   sheets: int, [sheets ≥ 0]
9 }
10 device PaperInputModule {
11   location tray1: UniformStack
12   location tray2: UniformStack
13   [size(output) == tray1.sheets + tray2.sheets]
14   [sum(output.map { sheet =>
15     if (
16       sheet.width == tray1.width and
17       sheet.height == tray1.height
18     )
19     1
20     else
21     0
22   }) ≥ tray1.sheets]
23   [sum(output.map { sheet =>
24     if (sheet.width == tray2.width and sheet.height == tray2.height)
25     1
26     else
27     0
28   }) ≥ tray2.sheets]
29   // A count function could make above more expressive
30   /*
31   [count(input, { sheet =>
32     sheet.width == tray1.width and
33     sheet.height == tray1.height
34   } >= tray1.sheets)]
35   [count(output, { sheet =>
36     sheet.width == tray2.width and
37     sheet.height == tray2.height
38   } >= tray2.sheets)]
39   */
40   location output: list<Sheet>
41 }

```

Figure 3.13: The CSX model resulting from iteration 3; the first attempt at modeling the input trays. In comments, it depicts how a count operator (which is not yet in CSX) could improve expressiveness. Note that this approach is incorrect for the case where the sheets in tray 1 and tray 2 are equal. Counterexample: both tray 1 and tray 2 contain one sheet with width 1, height 1, weight 1. The output stack could contain a sheet with width 1, height 1, weight 1 and a random second sheet, and still meet the constraints.

```

1 type UniformStack {
2   sheet: Sheet,
3   count: int, [count ≥ 0]
4 }
5
6 enum Tray { A B }
7
8 type Sheet {
9   width: int,
10  height: int,
11  isPortrait = height ≥ width
12 }
13
14 type Stack {
15   sheets: list<Sheet>,
16   stitches: list<Stitch>
17 }
18
19 type Stitch {
20   e: edge,
21   direction: Direction
22 }
23
24 enum Direction { Upwards Downwards }

```

Figure 3.14: The type definitions accompanying the final CSX model (Figure 3.15).

stack should contain the sheets defined in tray 1 and those in tray 2, in any order. The counting is modeled by combining a map to a list of zeros and ones and then a sum. A count operator would help to express this (see commented part in Figure 3.13).

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

In iteration 6, the participants re-modeled the tray assignment. This approach was also included in the final model (Figure 3.15). In this new approach, the participants included an enum with values for each sheet and added a list that indicates the tray assignments for each sheet. By doing so, each sheet is actually from one of the trays – if a sheet gets tray A assigned, its value in the stack must be equal to the sheet defining the uniform stack in tray A. Additionally, the number of assignments per sheet is counted and compared to the number of sheets in the uniform stacks of the trays. This ensures that the total number of sheets adds up. The participants observed that this was


```

1  device PaperInputModuleAndStitcher {
2      location entryA: UniformStack
3      [entryA.sheet.width ≤ entryA.sheet.height]
4      location entryB: UniformStack
5      [entryB.sheet.width ≤ entryB.sheet.height]
6
7      parameter oA: orientation [oA == rot0 or oA == rot90]
8      parameter oB: orientation [oB == rot0 or oB == rot90]
9
10     [oA == rot0 implies entryA.sheet.width == trayA.sheet.width and
11     entryA.sheet.height == trayA.sheet.height]
12     [oA == rot90 implies entryA.sheet.width == trayA.sheet.height and
13     entryA.sheet.height == trayA.sheet.width ]
14     [oB == rot0 implies entryB.sheet.width == trayB.sheet.width and
15     entryB.sheet.height == trayB.sheet.height]
16     [oB == rot90 implies entryB.sheet.width == trayB.sheet.height and
17     entryB.sheet.height == trayB.sheet.width ]
18
19     location trayA: UniformStack
20     location trayB: UniformStack
21
22     [entryA.count == trayA.count]
23     [entryB.count == trayB.count]
24     [trayA.count ≥ trayB.count]
25
26     location assignment : list<Tray> [size(assignment) == size(input)]
27     location input: list<Sheet> [size(input) == trayA.count + trayB.count]
28
29     [input.forall { sheet =>
30         if (sheet == trayA.sheet)
31             assignment[index] == A
32         else
33             (sheet == trayB.sheet and assignment[index] == B)
34     }]
35
36     var xA: list<int> var xB: list<int>
37
38     [size(xA) == size(assignment) and assignment.forall { x =>
39         xA[ index ] == (if (x == A) 1 else 0)
40     }]
41     [sum(xA) == trayA.count]
42
43     [size(xB) == size(assignment) and assignment.forall { x =>
44         xB[ index ] == (if (x == B) 1 else 0)
45     }]
46     [sum(xB) == trayB.count]

```

Figure 3.15: The final CSX model resulting from the co-design sessions (continued on next page). It integrates the key aspects of iteration 6 (properly modeling the input trays) and iteration 5 (edge stitching and orientations).

```

1 // Max size
2 [input.forall { sheet => sheet.width ≤ 4500 and sheet.height ≤ 3300}]
3 // Min size
4 [input.forall { sheet => sheet.width ≥ 1200 and sheet.height ≥ 1500}]
5
6 [size(input) ≤ 50] // Max number of sheets that can be stitched
7
8 [input.forall { sheet => sheet.height == first(input).height }]
9
10 [gathered.sheets == reverse(input)] // Gathering a sequence of sheets
    will have the first sheet at the bottom of the stack
11
12 location gathered: Stack [size(gathered.stitches) == 0]
    [output.sheets == gathered.sheets]
13
14 [size(output.stitches) == 0 or size(output.stitches) == 2]
15 [output.stitches.forall { stitch => stitch.e == right and
    stitch.direction == Upwards }]
16
17 location output: Stack
18
19 parameter o2: orientation [o2 == rot0 or o2 == rot90]
20 [(o2 == rot0) implies output.sheets.forall {
21   sheet => sheet.width == reader.sheets[index].width and sheet.height
    == reader.sheets[index].height
22 }]
23 [(o2 == rot90) implies output.sheets.forall {
24   sheet => sheet.width == reader.sheets[index].height and
    sheet.height == reader.sheets[index].width
25 }]
26 [reader.stitches.forall { stitch => output.stitches[index].e ==
    orientate(stitch.e, o2) }]
27 [output.stitches.forall { stitch => stitch.direction ==
    reader.stitches[index].direction }]
28
29 [size(output.stitches) == size(reader.stitches)] [size(output.sheets)
    == size(reader.sheets)]
30
31 location reader: Stack
32 }

```

Figure 3.15: The final CSX model resulting from the co-design sessions (continued).

challenging to implement due to the constraint-based paradigm of CSX, as it failed on the first attempt and required additional data structures and extensive testing to get right (*GO 11*).

Sheets must have equal height. In iteration 4, the participants included the constraint that all sheets must have the same height, which was also included in the final model (Figure 3.15, line 36). This captures a physical limitation of the device. The participants modeled this limitation using a forall that enforces all heights of the sheets to equal the height of the first sheet.

Edge Stitching. In iteration 5, the participants included the stitching capabilities of the device, which was also included in the final model (Figure 3.15). Again, for modeling the stitches, the first question that came to mind for the participants was which level of detail to include. The participants started with modeling stitches with an edge and a direction (Figure 3.14, lines 20–21). The *e* property has type *edge*, i.e., one of the geometrical constructs introduced in this chapter. A stitch can be applied in the upward or downward direction, which is modeled using an enum.

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

The device can apply multiple stitches to the stack of sheets, and therefore the participants used a list of stitches to model this. In addition to the usage of lists for modeling non-uniform stacks of sheets, the list construct in CSX is useful for coverage of a variable number of stitches (*COVERAGE 7*).

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

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

stitching, and we used it to model the edge of the stack on which the stitching occurs.

Orientations. In iteration 5, the participants modeled the aspect of orientations, which was also included in the final model (Figure 3.15). The device has the capability of orientating the stack of sheets before and after applying the stitches. This is useful because then a stack of sheets that is too large to be stitched in portrait orientation (lines 31–32) can be stitched in landscape orientation but still be presented in portrait orientation to the reader.

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

The capability of our device to orientate the stack of sheets has interaction with the geometrical constructs such as the edge on which a stitch is applied. The edge construct in CSX — including built-in transformations with orientations — makes it easy to express the transformation of an edge on a sheet (Figure 3.15, line 55). However, the participants noticed that such transformations on sizes are not built into CSX (Figure 3.15, lines 48–53). For example, when rotating a sheet by 90 degrees, the width and height are swapped. Although we can still express this in CSX with low-level modeling, CSX models would benefit from also having transformations of sizes expressed similar to edges (*COVERAGE 10*).

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

Positive observations:

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

Negative observations:

- CSX does not cover precision and units: the modeler needs to choose a precision, and configurations that are found need to be interpreted under the chosen precision (*COVERAGE 1*).

- Object terms, e.g., in tests, cannot be specified in terms of the domain, and need to be specified using low-level properties (*COVERAGE 4*).
- Handling grouping and ordering of sheets and stacks is not specifically covered in CSX and thus remains cumbersome to model (*COVERAGE 6*).
- The set of geometrical constructs in CSX is not complete and should be extended with directions (*COVERAGE 8*) and sizes (*COVERAGE 10*), because currently they require low-level modeling (*COVERAGE 9*).

3.4.2 Configuration Accuracy

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

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

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

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

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

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

Table 3.3: Scenarios of configuration space exploration for the final model of the coverage study (Figure 3.15) that we use for benchmarking. The Di scenarios *derive* a configuration. The Oj scenarios find an *optimal* configuration by either minimizing or maximizing an objective. All scenarios use 10 as the upper bound on list sizes.

ID	Description
D1	Output landscape A ₃ with right edge stitch
D2	Output portrait A ₃ with stitches (which then need to be on the top edge)
D3	Portrait A ₃ with stitch on right edge (which is not possible)
O1	Derive smallest portrait size with right edge stitch
O2	Derive smallest landscape size with right edge stitch
O3	Derive smallest portrait size with top edge stitch
O4	Derive smallest landscape size with top edge stitch
O5	Derive largest portrait size with right edge stitch
O6	Derive largest landscape size with right edge stitch
O7	Derive largest portrait size with top edge stitch
O8	Derive largest landscape size with top edge stitch

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

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

3.4.3 Configuration Performance

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

Although the performance of constraint solving is hard to predict in general, we conduct experiments to get an idea of the typical response times for typical configuration scenarios at Canon Production Printing. In particular, we take the final model of the domain coverage study and we define realistic scenarios of configuration space exploration for it. Table 3.3 lists the scenarios that we consider, which includes three scenarios that *derive* a configuration (including one for which no configuration can be found) and eight scenarios that find

an *optimal* solution. All scenarios use an upper bound on lists of 10 and all consider an output stack with five sheets.

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

To get an impression of the impact of list sizes on performance, we repeatedly benchmark the first scenario for multiple list upper bounds. For that, we alter scenario D1 such that the output stack size that is considered is half of the upper bound on lists. This ensures that the lists in the MiniZinc solution both have relevant values (that are considered in the configuration) and framed values (which are ignored). For example, for the test with an upper bound of list sizes of 300, the scenario considers an output stack of 150 sheets. For these benchmarks, we do not set a timeout.

We use the JMH framework¹ to implement the benchmarks, which is a framework for benchmarks in Java. Spoofox offers a core library that allows us to integrate the relevant components of CSX in the benchmark such that we can measure the translation time and solving time separately. We executed the benchmarks on a laptop with a quad-core processor with a base frequency of 3.1GHz and 16GB RAM, running macOS 12.4 and using Java version 1.8.0_275. Furthermore, we used version 2.6.4 of MiniZinc with two common solvers [41]: Gecode² and ORTools³. For each scenario, we first run 10 warmup iterations, then run 10 measurement iterations, and we report the average of the measurement iterations. We only report the results of the best-performing solvers, based on the least timeouts.

Results. The Gecode solver completed the benchmarks with the least timeouts, and therefore we only report the Gecode benchmark results. Figure 3.16 depicts the benchmarking results for the Gecode solver for all scenarios. The results show that most of the scenarios succeed within one second, and thus stay within the order of seconds time limit (*PERFORMANCE 1*). Two of the optimization scenarios (D4 and D8) — although they seem comparable to the other optimization scenarios — timed out (*PERFORMANCE 2*). For all scenarios, the translation times are higher than the solving times. The ORTools solver on average performed better on the derivation scenarios, but it timed out on all optimization scenarios.

Figure 3.17 depicts the benchmarking results for increasing the list upper bounds on scenario D1. The results indicate that increasing the list's upper bounds negatively impacts performance (*PERFORMANCE 3*). This is expected, as increasing the list's upper bound increases the solution space in which solvers need to find a solution. It is unclear yet for which cases in practice this

¹<https://openjdk.java.net/projects/code-tools/jmh/>

²<https://www.gecode.org>

³<https://developers.google.com/optimization>

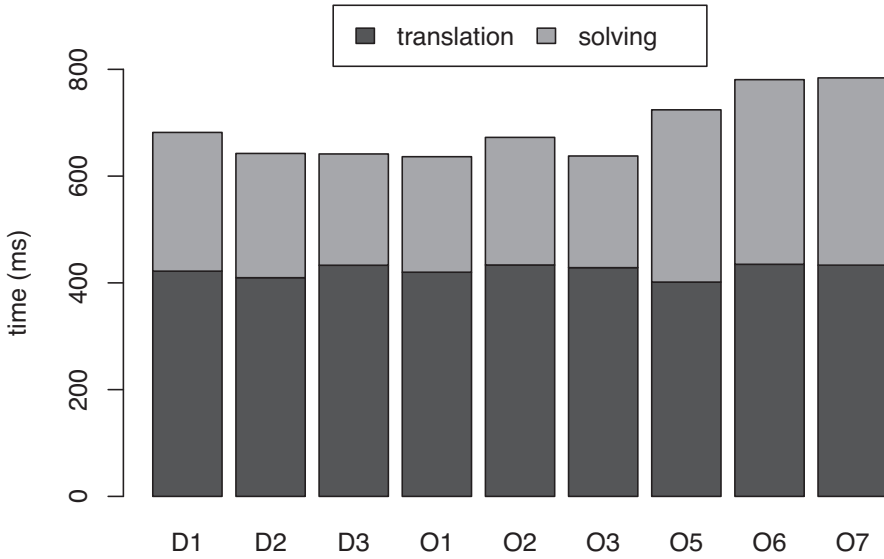


Figure 3.16: Benchmarking results for the Gecode solver for the scenarios from Figure 3.3. The bars show the translation and solving time separately. Times are reported in milliseconds. The tests D₄ and D₈ timed out and therefore are not included in the figure.

could become problematic.

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

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

3.4.4 Relevance

Study Setup. To evaluate the relevance of CSX 2.0, we gather anecdotal evidence by interviewing the domain expert and by considering general observations from the coverage study (Section 3.4.1). In particular, for the relevance of CSX 2.0 for developing control software for printing systems, we consider sufficiency and necessity:

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

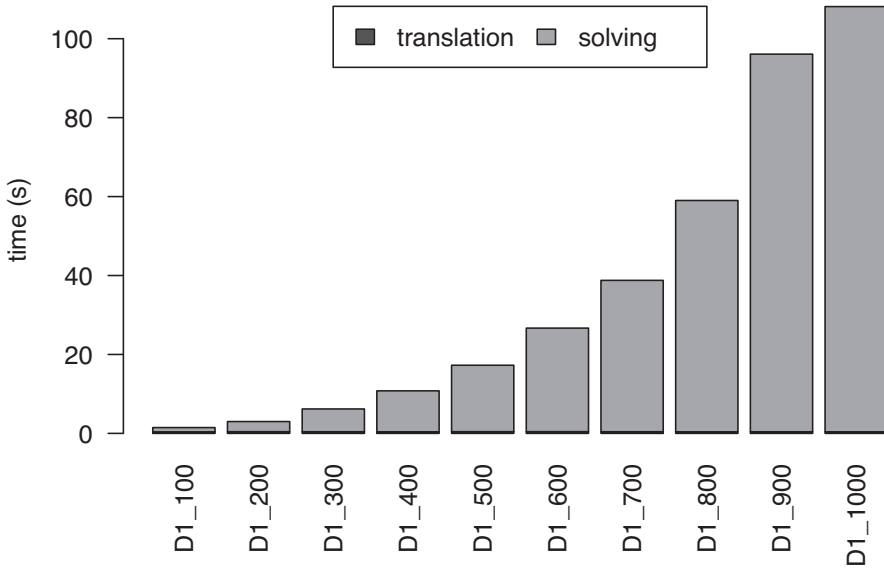


Figure 3.17: Benchmarking results for the Gecode solver for scenario D1 with list upper bounds varying from 100 to 1000 in which the output stack size is half of the upper bound on lists.

pre-CSX situation?

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

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

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

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

The domain expert reports that a key change in CSX with respect to pre-CSX

is the language's declarative nature. With CSX, modeling the printing system only concerns thinking about the characteristics of devices, and not about *how* to compute or find configurations for the devices. Given a CSX model, the configuration space exploration becomes an independent concern that can be fully automated (*RELEVANCE 3*).

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

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

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

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

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

The domain expert also mentions the level of abstraction as a key char-

acteristic that makes CSX more realistic to use in practice than MiniZinc (*RELEVANCE 7*). The domain expert reports that CSX is capable of abstracting over the complexity of low-level constraint modeling, by offering high-level language constructs.

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

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

- CSX is relevant because it realizes configuration space exploration (*RELEVANCE 2*) that is automatic (*RELEVANCE 1*) by only modeling device characteristics (*RELEVANCE 3*) and without requiring repeating development of algorithms for new devices (*RELEVANCE 4*).
- CSX is relevant because it has the potential to increase control software development productivity by greatly reducing development and validation time (*RELEVANCE 5*).
- CSX is relevant because, in contrast to a generic constraint modeling such as MiniZinc, the language includes constructs specific to the printing domain (*RELEVANCE 6*) which are on a higher level of abstraction (*RELEVANCE 7*), accompanied with an IDE with useful features such as inhabitation checks and interactive testing (*RELEVANCE 8*).

3.5 DISCUSSION

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

3.5.1 Language Design

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

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

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

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

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

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

Object Constructors. In Figure 3.18b we depict how the modeling of objects could be improved in a next version of CSX. By introducing object constructors, the test object can be specified in terms of user-defined types. If a property of the object is not relevant, it can be ignored by using a wildcard, which means the property could get any value. We expect extending CSX with support for object constructors with wildcards to be relatively straightforward.

Sizes. In Figure 3.19, we compare modeling the transformation of sheet sizes in CSX 2.0 (similar to in our evaluation case) with an alternative approach in a hypothetical CSX 3.0. By extending the set of geometrical constructs in CSX with sizes, size transformations can be modeled without having to model independent cases. Additionally, by using object constructors, a map operator can express a change over a list of items by conveniently modeling which properties do and which do not change.

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

```

1 type Sheet {
2   width: int,
3   height: int,
4   thickness: int
5 }
6 device MyDevice {
7   location in: Sheet
8   ...
9 }
10 test device MyDevice
11 config in.width = 210
12 config in.height = 297 {
13   ...
14 }

```

(a) CSX 2.0: individually specified properties.

```

1 type Sheet {
2   width: int,
3   height: int,
4   thickness: int
5 }
6 device MyDevice {
7   location in: Sheet
8   ...
9 }
10 test device MyDevice
11 config in = Sheet(210, 297, _) {
12   ...
13 }
14 }

```

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

Figure 3.18: The partial model of a sheet object in a test in CSX 2.0 and hypothetical CSX 3.0.

Figure 3.20 demonstrates this.

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

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

CSX 2.0 supports a single point of configuration for list upper bounds (GO 10). If it is known that a list will have a small maximum size, e.g., for a device that can only stitch 6 stitches maximum, it would be a better and more efficient model of the solution space if the instance of a specific list could get its own upper bound. We expect extending CSX with upper bounds for lists that are configurable per list instance to be relatively straightforward.

Reuse. In the coverage study, the participants did not yet make use of the actions language construct (Section 3.2.3) (GO 6). We expect the reason for this to be that actions are useful for factoring out common pieces of behavior (for which they were intended), but that it only becomes useful when a wider range of devices are modeled. To get a better understanding of the usefulness of actions in capturing reusable parts of printing behavior, we need a study on more devices. For a library of actions to be useful in practice, we think it is

```

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

```

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

```

1  enum Color { White Red }
2  type Sheet {
3    size: size,
4    color: Color
5  }
6  device Rotator {
7    location input: list<Sheet>
8    parameter o : orientation
9
10   [output == input.map {
11     sheet => Sheet(orientate(sheet.size, o), sheet.color)
12   }]
13
14   location output: list<Sheet>
15 }

```

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

Figure 3.19: Modeling the transformation of sheet sizes in CSX 2.0 (low-level) and in hypothetical CSX 3.0 (high-level).

```

1 type Sheet {
2   width: int, height: int
3 }
4 type Stack {
5   sheet1: Sheet,
6   sheet2: Sheet,
7   ...
8   sheet5: Sheet,
9   size: int,
10  [0 ≤ size and size ≤ 5]
11 }

```

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

```

1 type Sheet {
2   width: int, height: int
3 }
4 type Stack {
5   sheets: list<Sheet>
6 }
7
8
9
10
11

```

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

Figure 3.20: Example type definitions for modeling a non-uniform stack of sheets in CSX 1.0 (with a workaround) and in CSX 2.0 (using the `list` construct).

necessary that CSX also supports importing (GO 7). This would enable that the library of actions can be defined separately and types and actions from the library can be imported into specific device models.

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

Modeling the input trays in the coverage study was considered challenging by the study participants (GO 11). In Figure 3.15, the code for modeling input trays is duplicated for trays a and b. Although this part could be factored out in an action to become reusable, it still would require redundant modeling for cases with more than two trays.

In the coverage study, the device was limited to rotating the stacks by 0 or 90 degrees. When the set of possible orientations is increased, the number of orientations cases that need to be handled such as in Figure 3.15 (lines 48–53) grows. Partially, this manual handling of orientations could be resolved by supporting sizes (COVERAGE 10) and object constructors (see Figure 3.19). However, when the orientations with flips are allowed, this is still not sufficient. When a stack is flipped, the order of the sheets also becomes reversed.

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

3.5.2 Constraint-Based Programming

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

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

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

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

Level of Detail and Solving Performance. In theory, one could go as far as modeling a sheet of paper as a set of atoms. In practice, that would not be feasible with respect to solving performance, and it also does not have practical utility. In our work, the question remains what the actual level of detail in a model needs to be. In general, modeling with CSX involves a tradeoff between including more detail on the one hand and improving performance on the other hand. Based on our current experiences, we cannot yet conclude if CSX would have performance that is good enough for integration in UIs for all printing devices.

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

improved as well.

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

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

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

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

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

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

Traceability. The current version of CSX only reports a single configuration for a requested (partial) configuration or job specification, or it reports that no configuration is possible for a job. If no configuration can be found, there is no further indication of *why* no configuration can be found. In practice, this would hinder the usability of the system for operators. Possibly, existing

approaches for identifying minimal unsatisfiable sets [56] could help in tackling this. Then, characterizations of minimal unsatisfiable sets should be mapped from the constraint level back to the CSX level to make them understandable for operators.

3.5.3 *Application in Practice*

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

Integration with Control Software and UI. CSX currently solves the problem of modeling devices and realizing automated configuration space exploration, but requires the realization of more of the components in the architecture of Figure 3.6 for application in practice. Realizing these components requires a substantial investment, but the potential software engineering productivity gains and added functionalities can compensate for that investment. The two most important components that currently are missing are the integration with a user interface and code generation for instructing low-level embedded software.

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

CSX aims to realize configuration space exploration that is automatic and to have an effective and scalable method for integrating a large range of finishing devices. The integration of a device comprises more than just the modeling of the configuration space. Infrastructure is needed to — for a given configuration — instruct low-level embedded software components to operate under a configuration. The pre-CSX software already tackles this concern and thus CSX can become a layer on top of pre-CSX, generating the low-level control software components.

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

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

Language Engineering. Using a DSL to develop software in a company introduces a dependency on language engineering. In our work, the use of

a language workbench has done much of the “heavy lifting”; Spoofox provided and automated a large part of the language infrastructure for free, by generating parsers, compilers, and an IDE from language specifications.

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

Besides the dependency on language engineering as a skill, our implementation of CSX in Spoofox also imposes a dependency on the Spoofox tool. Although we found that Spoofox was effective for the implementation of CSX, we think other state-of-the-art language workbenches [7, 57, 8] could be used as well. If another tool for language development becomes preferred, the CSX language implementation could be ported. CSX has textual syntax, which eases the migration to another tool as the grammar can be ported, and existing CSX models can be maintained. With visual syntax or projectional editing this migration could be less straightforward.

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

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

3.5.4 *Lessons Learned*

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

1. The Spoofox language workbench and the MiniZinc constraint modeling language (and compatible solvers) took care of much of the “heavy lifting” in realizing CSX. This enabled us to tackle complexity and improve functionality in software engineering for a complex domain by allowing us to mostly focus on the domain and language design.
2. A systematic approach to DSL evaluation is useful for communicating about a DSL in an industrial context. Concrete evaluation criteria for the use of a DSL help in the discussion to explain to people who have no experience with DSLs to understand what is required for a DSL to be applied in practice. Finally, the evaluation criteria guide decision-making regarding the adoption of the technique.
3. Starting to use a DSL in practice has a big impact on the software engineering process with dependencies on external tooling and having language engineering resources available for both language development and language maintenance. Therefore, the benefits of adopting a DSL need to be large to outweigh the corresponding investment.
4. The conceptual power of CSX is amplified by its IDE. The CSX IDE gives helpful insight into the behavior of models by featuring interactive validation of tests and debugging through inspection of configurations. This helped us to try out alternative language designs, leading to an iterative language design process.
5. It is a crucial language design decision to have types being defined in a language itself — instead of embedding a fixed set of domain objects in the language — which enables flexibility in modeling by iteratively including more detail in models. In CSX, this enabled experimenting with different representations of objects from the printing domain without changing CSX itself.
6. A high level of abstraction and domain-specific constructs such as in CSX are necessary to make constraint-based modeling accessible. Still, switching to the constraint-based programming paradigm can be challenging for developers who have no experience with constraint programming or with declarative programming at all.

3.5.5 *Threats to Validity*

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

External Validity. We have presented an experience report that focuses on a particular industrial context, and therefore we do not claim that our findings are generalizable. Still, we think the outcomes of our work can be useful to others working in an industrial context where a domain-specific interface to constraint solving is useful. Ultimately, we need to further apply CSX on a

wider range of printing systems and with more engineers and domain experts to get a better understanding of the effectiveness and scalability of the method.

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

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

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

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

3.6 RELATED WORK

We describe related work in which high-level modeling languages interface with constraint solvers in the backend. We focus on more general constraint-solving approaches as our objective made us select SMT constraint solvers for CSX and because our practical experience showed that applying CSX involves models with various types of constraints (linear, logical, satisfiability). Whereas other work focuses on evaluating the tools used to create DSLs [57, 6], we focus on evaluating the DSL itself.

The work of Keshishzadeh et al. applies constraint solving in the backend of a DSL for the domain of medical imaging equipment [47]. In particular, they use constraint solving to validate domain-specific properties for realizing collision prevention in the equipment. If such properties are violated, the causes of violations can be traced through delta debugging and reported back on the model level.

KernelF by Voelter et al. is a reusable functional language for the modular development of DSLs [48]. KernelF features advanced error checking and verification based on constraint solving with the Z3 solver. In a case study on payroll calculations [49], these techniques are applied to statically check the

completeness and overlap of domain-specific switch-like expressions. These forms of static analysis are similar to the interactive analysis of CSX.

Although in this chapter we have focussed on CSX as a method to realize automatic configuration space exploration, the language also has the potential to cope with the large variety of finishers. The use of constraint solving is common in product line engineering, and, e.g., is also used in feature models of printing systems [17], but constraint solving in that context has a different utility than in CSX. Feature models can be used to model systems as compatible compositions of features or components, and constraint solving can be used to find or check feature compositions. CSX, in contrast, is used to find configurations at run time for a particular device.

De Roo et al. [58] present an architectural framework for realizing multi-objective optimization for embedded control software. Additionally, they introduce a toolchain that consists of visual editors, analysis tools, code generators, and weavers. The approach is based on domain-specific models from which optimization code is generated automatically. Both CSX and their work use constraint models for realizing control software and support solving for optimization objectives. The authors evaluate their work in the context of the industrial printing domain as well. Roo's DSL is targeted at a different sub-domain of printing software, namely embedded online control. Our domain represents the configuration spaces of a product family of hardware devices, and the configuration control software that can be derived from it. Our work on CSX is different in the sense that it is used before the execution of print jobs (offline) to derive configurations, whereas de Roo et al. focus on optimization in embedded control software that runs during the execution of print jobs (online), imposing different requirements. Finally, the aim of CSX is to involve domain experts such as mechanical engineers in the modeling process.

Constraint solving is also used in model checking and relational model finders. For example, Alloy [50] is a high-level specification language that features finite model finding to check formal specifications. Alloy uses KodKod [51], which is a relational model finder on problems expressed in first-order logic, relational algebra, and transitive closures. KodKod differs from CSX in several ways. In KodKod the nature of models is relational, where CSX considers fixed manufacturing paths and models the objects and parameters in such paths. KodKod does not support reasoning over data or optimization, whereas CSX does support optimization.

Stoel et al. extend relational model finding with first-class data attributes and optimization in AlleAlle [52]. Similar to CSX, AlleAlle includes data into problem models and uses SMT constraint solving for model finding. CSX and AlleAlle differ in the sense that AlleAlle is an intermediate language that targets relational problems, while CSX is a DSL specific to the printing domain and without first-class support for relations. AlleAlle and CSX both lack an approach for mapping reasons for unsatisfiability that are found on the constraint level back to the model level.

Muli [54] integrates constraint solving with the object-oriented programming paradigm by extending the Java programming language. Muli adds support

for symbolic values to Java, which translate to constraint variables in the runtime. Muli features a runtime that integrates constraint solvers in a Java virtual machine. In contrast to CSX, Muli is a general-purpose programming language, and it does not support lists or optimization.

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

3.7 CONCLUSIONS

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

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

3.7.1 *Future work.*

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

OIL: an Industrial Case Study in Language Engineering with Spoofox

ABSTRACT

Domain-Specific Languages (DSLs) promise to improve the software engineering process, e.g., by reducing software development and maintenance effort and by improving communication, and are therefore seeing increased use in industry. To support the creation and deployment of DSLs, language workbenches have been developed. However, little is published about the actual added value of a language workbench in an industrial setting, compared to not using a language workbench. In this chapter, we evaluate the productivity of using the Spoofox language workbench by comparing two implementations of an industrial DSL, one in Spoofox and one in Python, that already existed before the evaluation. The subject is the Open Interaction Language (OIL): a complex DSL for implementing control software with requirements imposed by its industrial context at Canon Production Printing. Our findings indicate that it is more productive to implement OIL using Spoofox compared to using Python, especially if editor services are desired. Although Spoofox was sufficient to implement OIL, we find that Spoofox should especially improve on non-functional aspects to increase its adoptability in industry.

Based on: Olav Bunte, Jasper Denkers, Louis van Gool, Jurgen J. Vinju, Eelco Visser, Tim Willemse, and Andy Zaidman. "OIL: an Industrial Case Study in Language Engineering with Spoofox". In: *Software and Systems Modeling* (2024). DOI: 10.1007/s10270-024-01185-x.

4.1 INTRODUCTION

Every piece of software is written in one or more software languages. The most common software languages are General-Purpose Languages (GPLs), such as C++, Java, and Python. For specific purposes, it can be beneficial to design a tailored language. Such a language is called a *Domain-Specific Language* [1] (DSL). Compared to GPLs, DSLs promise to improve the software engineering process, e.g., by reducing development and maintenance effort when implementing (domain-specific) software. They are also considered to be more suitable for communication between software engineers and domain experts [59].

To support the creation and deployment of DSLs, *language workbenches* have been developed [60, 7, 8]. Language workbenches are specifically designed for the development of a DSL. This includes the DSL's syntax, from which parsers can be derived automatically, as well as its semantics, e.g., by means of a translation to other languages. Language workbenches typically also generate IDEs for the DSLs implemented in them. Examples of language workbenches are MPS [61], Xtext [62], Rascal [23], and Spoofox [20].

Although there already is ample literature on the underlying theory of language workbenches (e.g., [8, 28]), little is documented about the actual added value of language workbenches compared to not using language workbenches in an industrial setting when designing and engineering DSLs. This is relevant for two main reasons. On the one hand there is opportunity: there exist DSL implementations in industry which have not been developed with the potential benefit of language workbenches. On the other hand there are still unknowns: most language workbenches spawn from academic environments which can have different views on software engineering effectiveness compared to a pure industrial setting. How relevant are the benefits of language workbenches in an industrial setting?

One of the first works that evaluates the added value of a specific language workbench in an industrial context is the work by Van den Brand et al. [63]. In this work, the authors present some experiences with using ASF+SDF for railway and financial domains. A more recent and extensive industrial case study is described in the work of Voelter et al. [6]. The authors evaluate the MPS language workbench with as case the mbeddr collection of languages under non-trivial requirements. The work by Voelter et al. resulted in meaningful lessons learned for the particular case study from an industrial perspective. Still, the authors call for more studies on language workbench evaluation to expand our knowledge of the usefulness of language workbenches for language engineering in general. This will help industrial language engineers decide when and how to use language workbenches.

We present such an evaluation of the Spoofox language workbench in an industrial setting. In the original work on Spoofox [20], the authors claim that Spoofox *“enables efficient, agile development of software languages with state-of-the-art IDE support based on concise, declarative specifications”*. From this can be derived that it should be more productive to implement a DSL with Spoofox compared to when not using a language workbench. Although it has been

demonstrated that Spoofox is able to deliver on its original promises for non-industrial greenfield situations, e.g., in the area of web programming [64, 65], or declarative data modeling [66, 67], it is unclear to what extent the original claims of Spoofox still hold for our industrial case. This leads to the following research question:

RQ: *How does the productivity of implementing an industrial language in Spoofox compare to the productivity when using a GPL and available libraries?*

Productivity is about the amount of effort needed to implement some functionality. As a proxy for effort we measure code volume, as it is the only information available to us in this study that relates to effort and can be measured objectively.

The industrial case with which we evaluate Spoofox is the Open Interaction Language (OIL), a textual language for modeling control software, developed at Canon Production Printing¹. Before the implementation of OIL in Spoofox was created, a design of OIL already existed based on XML, along with an implementation in Python. This makes OIL a typical industrial case, in the sense that the new implementation must fit into an existing software ecosystem which is used to create commercial products.

The industrial context requires a number of features for the implementation of OIL. In particular, with the migration to Spoofox, the original XML syntax should be supported alongside a new more user-friendly syntax. OIL's syntax allows the user to leave out boilerplate information, which the implementation needs to make explicit. The well-formedness, name binding, and typing of an OIL specification should be statically checked and errors should be reported to the user. OIL specifications depend on modules and interfaces defined in another language called Interface Definition Language (IDL). Finally, the Spoofox implementation of OIL should be able to generate code, both for the execution and the verification of OIL specifications.

Based on the aforementioned requirements and earlier non-industrial evaluations of Spoofox, in this chapter we evaluate how well Spoofox can cope with the complexity and scale of the industrial OIL case study. The development of OIL in Spoofox, executed by five developers over more than four years, allows us to make interesting observations on language engineering, distill strengths and weaknesses of Spoofox, derive lessons learned for future language engineering efforts, and propose areas of future work to improve the language workbench.

Outline. This chapter is structured as follows. First, we provide background on Spoofox in Section 4.2 and on OIL in Section 4.3. We dive into the context, research method, and setup of the case study in Section 4.4. Next, we discuss the language engineering aspects of OIL's implementation in Spoofox in separate sections, and we evaluate our research question for each aspect at the end of those sections. We discuss the implementation of OIL's concrete syntax

¹<https://cpp.canon>

in Section 4.5. In Section 4.6 we discuss the abstract syntax representation of OIL. Then in Sections 4.7 and 4.8 we discuss the implementation of the static and dynamic semantics of OIL, respectively. In Section 4.9 we summarize our findings regarding the research question and discuss threats to validity. We discuss experiences that are not directly related to the research question in Section 4.10, as well as list our lessons learned and provide a research agenda for Spoofox. We position our work with that of others in Section 4.11. We conclude in Section 4.12.

4.2 SPOOFAX

In this section, we provide background information on Spoofox, which is useful for understanding the way that OIL is implemented in Spoofox in later sections. Spoofox² is an open source language workbench that promises to support the development of textual DSLs by offering meta-DSLs (DSLs for developing DSLs) for concise, declarative specifications of languages and IDE services [31]. The idea of declarative language definition is that language developers focus on the high-level specification of their languages rather than focusing on the low-level implementation of, e.g., parsing or type-checking algorithms. Based on language aspects specifications in the meta-DSLs, Spoofox automatically generates an IDE.

Spoofox is developed at the Delft University of Technology since 2007 [20], building on previous work on syntax definition with SDF2 [68] and program transformation with the Stratego XT toolset [43]. Besides SDF2 and Stratego, the first version of Spoofox offered meta-DSLs for static semantics (NaBL2 [45]), editor services (ESV), and testing (SPT). Spoofox Core [69], implemented in Java, integrates the meta-DSLs and provides a build system to automatically transform language specifications into implementations. Spoofox is primarily deployed as a plugin for the Eclipse IDE³.

Developments on Spoofox since the introduction of the language workbench include:

- **Syntax.** The syntax definition formalism SDF3 [42] with support for template-based syntax definition.
- **Transformation.** The program transformation language Stratego 2 with support for gradual typing [70] and incremental builds [71].
- **Static Semantics.** Static semantics specification (NaBL2 and its successor Statix [72], both based on a scope graph model [44]) and support for incremental type checking [73].
- **Data-Flow.** The data-flow analysis specification language FlowSpec [74].
- **Incremental Builds.** Interactive software development pipelines with PIE [75].
- **IDE support.** Static semantic code completion [76].
- **Testing** Language test suites with the Spoofox Testing language (SPT).

²<https://spoofox.dev>

³<https://www.eclipse.org/ide/>

```
let x = 20 + 1 in 2 * x
```

```
Exp(  
  Let(  
    "x"  
    , Add(Int("20"), Int("1"))  
    , Mul(Int("2"), Ref("x"))  
  )  
)
```

Figure 4.1: An example program in EXP and its corresponding abstract syntax in ATerm.

The case study in this chapter has been performed with Spoofox version 2 [69], making use of the SDF₃, NaBL₂, Stratego, ESV, and SPT meta-DSLs. Spoofox version 3 (including Stratego 2, Statix, and PIE) was under development during the execution of this study, and could therefore not yet be considered. In Section 4.10, we discuss how our findings relate to Spoofox 3.

In the remainder of this section, we discuss both conceptual and practical aspects of language engineering with Spoofox, which are important for understanding the Spoofox implementation of OIL. Also, we will further introduce the meta-DSLs for the language aspects that are relevant in our case study. We use a simple expressions language EXP as a running example, which supports integers, addition, multiplication, let bindings and references. Fig. 4.1 depicts an example EXP program and its corresponding abstract syntax.

4.2.1 Anatomy of Spoofox Projects

A Spoofox project consists of source files and configuration files. The source files primarily consist of specifications in the meta-DSLs. For integrating a language implementation with external libraries, Java source files can be included as well. The language build and dependencies are configured in the configuration files. All sources files are textual and are therefore typically stored in a version control system.

Based on the source files and configuration files, Spoofox generates language artifacts such as parse tables, AST schemas, and ultimately the complete language implementation in the form of an Eclipse plugin. During a language build, besides the sources that the language developer writes, additional sources are generated which can be referenced by other specifications or form an input for a further build step. For example, signatures are generated automatically from the SDF₃ grammar and can be used in Stratego to define transformation rules on. These generated sources are stored separately from the main source files and are typically ignored in version control.

Projects can define a complete language, define (library) sources intended for reuse by other language projects, or only define a transformation for an existing language. Through dependencies between projects, different forms of

```
signature
constructors

Int : INT -> Exp
Add : Exp * Exp -> Exp
Mul : Exp * Exp -> Exp
Let : ID * Exp * Exp -> Exp
Ref : ID -> Exp
```

Figure 4.2: A Stratego code snippet that defines an AST schema for EXP.

language composition can be realized. For example, a language project can re-use definitions of another language by adding that project as a dependency. Also, a project can contribute a transformation to an existing language, such that more functionalities become available for a language, independent from its original implementation.

4.2.2 Data Representation with ATerms

The language ATerm (Annotated Terms) [77] defines the representation of abstract syntax trees (ASTs) and data used by most other meta-DSLs. These ASTs and data consist of terms (often referred to as “ATerms”) that can be annotated with additional data. A term can either be a number, a string, a list of terms, or a constructor with zero or more subterms. The annotations on terms are typically used to store metadata, such as static analysis results. ATerm serves as the “glue” between the meta-DSLs. For example, the output of an SDF₃-based parser is an AST expressed in ATerm. ATerm is the object language for the Stratego transformation meta-DSL, i.e., Stratego defines transformation rules for terms expressed in ATerm. Also, name binding and typing specifications in NaBL₂ consist of rules that apply to ATerm patterns.

Terms must adhere to many-sorted algebraic *signature* [78] definitions which are defined in Stratego. The signatures define *sorts* and *constructors*. Sorts represent syntactic categories (also known as non-terminals, e.g., `Exp`) and constructors specify instances of these sorts (e.g., `Add`). The snippet in Fig. 4.2 contains signature definitions for EXP. It defines unary `Int` and `Ref` constructors for the sort `Exp`, binary `Add` and `Mul` constructors for the sort `Exp`, and a ternary `Let` constructor. Fig. 4.1 contains an example term that conforms to the signatures.

4.2.3 Syntax Definition with SDF₃

SDF₃ [42] is a syntax definition language that covers more than realizing a parser implementation based on a grammar specification. From an SDF₃ grammar, the following language implementation artifacts are generated automatically: an AST schema, a parser with error recovery, a pretty printer, a parenthesizer, syntax highlighting, and syntactic code completion. The SDF₃ formalism extends context-free grammars [79] with high-level syntax

definition features such as constructor declarations (used for AST schema generation in the form of ATerm signatures), disambiguation constructs (for disambiguation and generating a parenthesizer), and templates (for deriving pretty printers) [80].

See Fig. 4.3 for two modules of SDF₃ that define the syntax of EXP. The second module (`exp`) imports the first module (`lex`). For example, in `exp`, the rule on line 7 defines a rule for integers, using the lexical syntax for INT defined in `lex`. The rules on lines 8-9 are defined to be left-associative using the `{left}` disambiguation construct. The left-hand sides of grammar rules consist of a sort (e.g., `Exp`) and optionally a constructor declaration (e.g., `.Add`). The signatures of Fig. 4.2 are generated automatically based on the constructor declarations in this grammar.

In addition to associativity declarations for disambiguation of an operator with itself, the `context-free priorities` section defines disambiguation through priorities between operators. In the example, `Exp.Mul` has higher priority than `Exp.Add` which has higher priority than `Exp.Let` (line 16). Priority declarations are transitive. When importing modules in SDF₃, their disambiguation rules are imported with them as well. Note that this may create new ambiguities between grammar elements of different modules, so additional disambiguation rules may need to be defined.

SDF₃-based parsing involves the process of *imploding*, i.e., transforming parse trees into ASTs. Only the nodes in the parse tree that are parsed based on grammar rules for which a constructor has been declared end up in the AST, which filters out irrelevant details of the concrete syntax such as white space and comments. This filtering is necessary because SDF₃ uses a scannerless parsing approach [68, 81], a foundational characteristic which makes SDF₃ grammars composable.

During parsing and imploding, the created AST terms are annotated with the origin location of the parsed syntactic element in the input program, which is the first step of *origin tracking* [46]. These origins can be maintained during transformations, which is useful for, e.g., error reporting.

To adapt the formatting of the text produced by the pretty printer that is generated from the SDF₃ grammar, one can enhance the SDF₃ grammar with templates. The example of Fig. 4.3 uses such templates for `Add`, `Mul` and `Let`, which is indicated by the square brackets that are placed around the right-hand sides of the grammar rules. Any formatting between these square brackets, including spaces, tabs, and newlines, will be used by the pretty printer. In case square brackets are part of the grammar definition, angular brackets can be used instead.

4.2.4 Static Semantics with NaBL₂

NaBL₂ [45, 82], pronounced as “enable two”, is a static semantics definition language which covers name binding and type systems based on the *scope graph* model [44]. Given an NaBL₂ specification, programs are transformed into constraints and a scope graph which captures the binding structure and typing of the program. Name resolution corresponds to finding a path in the graph

```

1 module lex
2
3 lexical syntax
4
5 INT    = "-"? [0-9]+
6 ID     = [a-zA-Z] [a-zA-Z0-9\]*
7 LAYOUT = [\ \t\n\r]

```

```

1 module exp
2
3 imports lex
4
5 context-free syntax
6
7 Exp.Int = INT
8 Exp.Add = [[Exp] + [Exp]] {left}
9 Exp.Mul = [[Exp] * [Exp]] {left}
10 Exp.Let =
11   [let [ID] = [Exp] in [Exp]] {non-assoc}
12 Exp.Ref = ID
13
14 context-free priorities
15
16 Exp.Mul > Exp.Add > Exp.Let

```

Figure 4.3: Two SDF₃ code snippets that define the syntax for EXP.

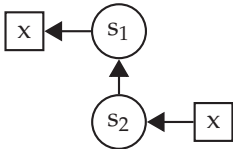


Figure 4.4: The scope graph that corresponds to the example EXP program of Fig. 4.1. Scope s_1 is the root scope node and corresponds to the whole program. Scope s_2 is the scope introduced for the body of the let operator, consisting of the expression $x = 20 + 1$. The declaration of x is represented by the outgoing edge from scope s_1 . The reference of x is represented by the incoming edge to scope s_2 . The name binding in this program is valid, as there exists a path from the reference of x to its declaration.

from a reference to its declaration. An NaBL2 specification contains constraint generation rules for every term in the AST schema of the language, with conditions that specify how the term contributes to scope graph generation, name binding, and typing.

Fig. 4.4 depicts the scope graph that corresponds to the example EXP program of Fig. 4.1. Each term is made part of a scope, which is a node in the scope graph. Declarations and references (e.g., of variables) are also added as nodes in the scope graph. For declarations in a scope, we add a node for the declaration with an edge from the scope to the declaration. For language constructs that introduce a deeper level in the overall scoping structure, the scope is added to the graph as a new node with an edge to the parent scope. For a reference, a node is added with an edge from the reference node to the node of the scope the reference is made from. Name resolution then boils down to finding a path in the scope graph from the reference to the corresponding declaration.

By assigning types to terms, type analysis can check or infer types. Conditions in constraint rules can be extended with an error message applied to a term. Whenever a condition fails, the error message can be displayed on the origin of the term using origin tracking.

See Fig. 4.5 for an NaBL2 snippet that specifies name and typing rules for EXP. Four rules are defined, identified in double square brackets (lines 2, 4, 9 and 17). The rule for term `Int` (line 2) assigns the type `TInt` to the term. The rule for `Add` recursively specifies the semantics for its sub-expressions by calling constraint rules on `exp1` and `exp2` using double square brackets (lines 5-6). Note that no rule references are used: the rule that needs to be applied depends on the outermost constructor of the sub-expressions. Afterwards it is defined that their types should be the same (line 7). For more complex type systems, it is possible to define relations between types. This enables, for instance, the addition of an integer with a float and the computation of the resulting type. The rule for `Mul` has been omitted as it is similar to the rule for `Add`.

The rule for `let` bindings (line 9) introduces a new scope `s'` (line 10), sets `s` as the parent scope of `s'` (line 11) and attaches a declaration node `Var{x}` for name `x` in the namespace `Var` to scope `s'` using an arrow pointing towards the declaration node (`<-`, line 12). It then analyses the first expression within scope `s` (line 13) and assigns the derived type `ty1` to the declaration node (line 14). Lastly, it continues the analysis with the second expression within scope `s'` (line 15). The rule for variable references (line 17) first attaches a reference node `Var{v}` for name `v` in the namespace `Var` on scope `s` using an arrow pointing away from the reference node (`->`, line 18). Afterwards it is checked whether some declaration `d` exists for reference `Var{v}` using operator `|->` (line 19), effectively checking whether there exists a path through the scope graph from the reference node to a declaration node with the same name and namespace. We then require that this declaration `d` has type `ty` (line 20), which is the same type as the `Ref` term that the rule is defined on (line 17).

```

1 rules
2   [[ Int(_) ^ (s) : TInt() ]].
3
4   [[ Add(exp1, exp2) ^ (s) : ty1 ]] :=
5     [[ exp1 ^ (s) : ty1 ]],
6     [[ exp2 ^ (s) : ty2 ]],
7     ty1 == ty2 | error $[Type mismatch: cannot add [ty2] to [ty1]].
8
9   [[ Let(x, exp1, exp2) ^ (s) : ty2 ]] :=
10    new s',
11    s' ---> s,
12    Var{x} <- s',
13    [[ exp1 ^ (s) : ty1 ]],
14    Var{x} : ty1,
15    [[ exp2 ^ (s') : ty2 ]].
16
17   [[ Ref(v) ^ (s) : ty ]] :=
18     Var{v} -> s,
19     Var{v} |-> d | error $[Cannot resolve [v]] @ v,
20     d : ty.

```

Figure 4.5: An NaBL2 code snippet that declares name binding and typing for EXP.

4.2.5 Transformation with Stratego

Stratego [83, 43] is a transformation language based on term rewriting and programmable rewriting strategies. Rewrite rules specify how a single input term transforms into an output term. These rules can be combined by putting them in sequence (e.g., $r1 ; r2$), by non-deterministically choosing between them (e.g., $r1 + r2$), or they can be passed to pre- or self-defined AST traversal strategies such as `topdown(r1)` or `bottomup(r1)`.

See Fig. 4.6 for an example Stratego transformation for EXP. Strategy `simplify0` simplifies expressions that contain zeroes by performing a bottom up traversal through the AST and trying to apply rule `simplify0-term` on every AST node. The rule `simplify0-term` is defined four times, each for a different type of expression that can be simplified. Each `simplify0-term` rule is tried until the AST node matches with the left hand side of the rule, after which the transformation is applied. The order in which the rules are tried is chosen non-deterministically during runtime. The `try` rule allows each `simplify0-term` rule to fail, which can happen for instance when the AST node is an `Int` term, after which it simply continues with the traversal.

Transformations with Stratego are generally model-to-model, which can be both endogenous (source and target are the same language) and exogenous (source and target are different languages) [84]. Fig. 4.6 is an example of an endogenous model-to-model transformation. It is also possible with Stratego to define model-to-text transformations, since a string is a valid term too.

```

strategies

  simplify0 = bottomup(try(simplify0-term))

rules

  simplify0-term: Add(Integer("0"), x) -> x
  simplify0-term: Add(x, Integer("0")) -> x
  simplify0-term: Mul(_, Integer("0")) -> Integer("0")
  simplify0-term: Mul(Integer("0"), _) -> Integer("0")

```

Figure 4.6: A Stratego code snippet that defines transformations on EXP for simplifying expressions.

```

strategies

  print-exp = bottomup(print-term)

rules

  print-term: Int(x) -> x
  print-term: Ref(v) -> v
  print-term: Add(x, y) -> $[[x] + [y]]
  print-term: Mul(x, y) -> $[[x] * [y]]
  print-term: Let(v, x, y) -> $[let [v] = [x] in [y]]

```

Figure 4.7: A Stratego code snippet that defines a printer for EXP.

```
menus
  menu: "Simplifications" (openeditor)
    action: "Simplify zeroes" = editor-simplify0
```

Figure 4.8: An ESV code snippet that adds an editor action to simplify expressions with zeroes in an EXP specification.

```
1 language EXP
2
3 test simplify addition with zero [[
4 3 * x + 0
5 ]] transform "Simplifications / Simplify zeroes" to EXP [[
6 3 * x
7 ]]
```

Figure 4.9: An SPT code snippet that defines a test for `simplify0`.

Stratego supports such transformations with *templates*, denoted with `$[. .]`. A template defines a string in which variables and transformation rules can be used to create substrings. See Fig. 4.7 for a transformation that prints an EXP AST. Note that given the syntax definition of EXP, such a pretty printer is generated automatically.

4.2.6 Editor Services with ESV

ESV is a language for defining editor services. An ESV specification can, e.g., customize syntax highlighting coloring and configure editor actions. See Fig. 4.8 for an ESV snippet that adds an editor action for EXP. This snippet defines a new menu called `Simplifications`, consisting of an action `Simplify zeroes`. This action is mapped to the transformation `editor-simplify0` (definition not explicitly shown), which applies `simplify0` to an EXP specification. These actions can be invoked in the IDE via the menu `Simplifications / Simplify zeroes` whenever an EXP file is in focus.

4.2.7 Testing with SPT

SPT [85] is a language testing framework for languages implemented in Spoofox. In SPT, test programs can be written and tested for errors and expected outputs. For testing static analysis, one can, e.g., provide an incorrect program where some elements have been marked. Then in the test expectation one can specify at which of these markers an error should occur. For testing transformations, one can provide an input program, an editor action to execute, and an expected output program. Such a test compares the AST that results from the editor action to the AST that results from parsing the expected specification, so the formatting of the provided specifications does not influence the test.

See Fig. 4.9 for an SPT snippet that defines a test for the `simplify0` transformation. Line 4 defines the input specification, line 5 defines the editor action to apply, and line 6 defines the expected output specification.

4.3 OIL

We first give an overview of OIL. Afterwards, we define a number of features that should be realized by the implementation of OIL in Spoofox.

4.3.1 History of OIL

OIL, which stands for Open Interaction Language, is a language developed by Van Gool (co-author) to model the behavior of control-software systems. In its early stages, OIL was designed to model the intended communication behavior between a group of components, known as a *protocol*. Later, OIL was adapted to also enable the modeling of individual components. Although OIL is developed at Canon Production Printing, it is not limited to modeling systems within the printing domain. The original syntax of OIL is XML based, but a more user-friendly DSL variant was created using Spoofox [36]. Though OIL is a textual language, it was designed to allow for an unambiguous visualization, as this is indispensable for communication between engineers.

With the development of OIL also came dedicated tooling. This tooling, implemented in Python, is able to parse and validate OIL specifications. It is a web-based environment in which OIL specifications can be inspected but not edited; editing happens inside a separate IDE, typically Visual Studio. The tooling also supports the visualization of OIL specifications, as well as simulation of traces over this visualization. For OIL component specifications it can generate executable code, which has been used to implement several complex software components for printers developed at Canon Production Printing. In this web-based tooling, OIL specifications can be pretty printed and editor services such as syntax highlighting and error reporting are available. In the rest of this document, we refer to this implementation of OIL as “the Python implementation”.

4.3.2 Overview of OIL

OIL is a state machine language that uses variables to store the current state. These variables and their values can be represented by areas, which are connected with transitions that can specify updates of variables, triggered by the occurrence of events. In this section, we give an informal description of the concepts of an OIL component specification and their semantics that are relevant for this chapter. For a more in depth description of OIL and a definition of its formal semantics, see [86].

We use the OIL component specification in Fig. 4.10 as running example. This OIL specification models a printer that, after a client has registered, can be turned on and off. When it is on, jobs of at most three sheets can be sent to the printer that are immediately processed. The printer also keeps track of its temperature and must be cooled down if it becomes too hot. See Fig. 4.11 for a

```

1 component heat2c {
2   import heat2ci
3   provides heat2ci.server
4   requires heat2ci.client
5
6   enum power {off, on}
7
8   var power : power
9   var client : heat2ci.client
10  var tmp : int32 = 20
11  var sheets : int32 = 0
12
13  state init
14  state active {
15    region power [this.power] {
16      state off ['off']
17      state on ['on']
18    }
19
20    zone power_on [this.power == 'on'] {
21      region job {
22        state idle
23        state busy
24      }
25    }
26
27    zone heat [this.tmp < 45]
28  }
29
30  concern REGISTRATION {
31    in init on register_client() assign this.client := client go active end
32  }
33
34  concern POWER {
35    in off on turn_on() go on end
36    in on on turn_off() go off end
37  }
38
39  concern JOB {
40    in idle on add_job(nrsheets) if nrsheets > 0 and nrsheets <= 3 assign
41      this.sheets := nrsheets go busy end
42    in busy if this.sheets == 0 do [silent] job_printed() go idle end
43    in busy if this.sheets > 0 at this.client do sheet_printed(sheetnr =
44      this.sheets) assign this.sheets := this.sheets - 1 go busy end
45  }
46
47  concern HEAT {
48    in heat on turn_on() assign this.tmp := this.tmp + 5 go heat end
49    in heat if this.tmp > 20 on cool_down() assign this.tmp := 20 go heat end
50  }
51 }

```

Figure 4.10: The OIL specification for an overheating printer (in the newer DSL notation).

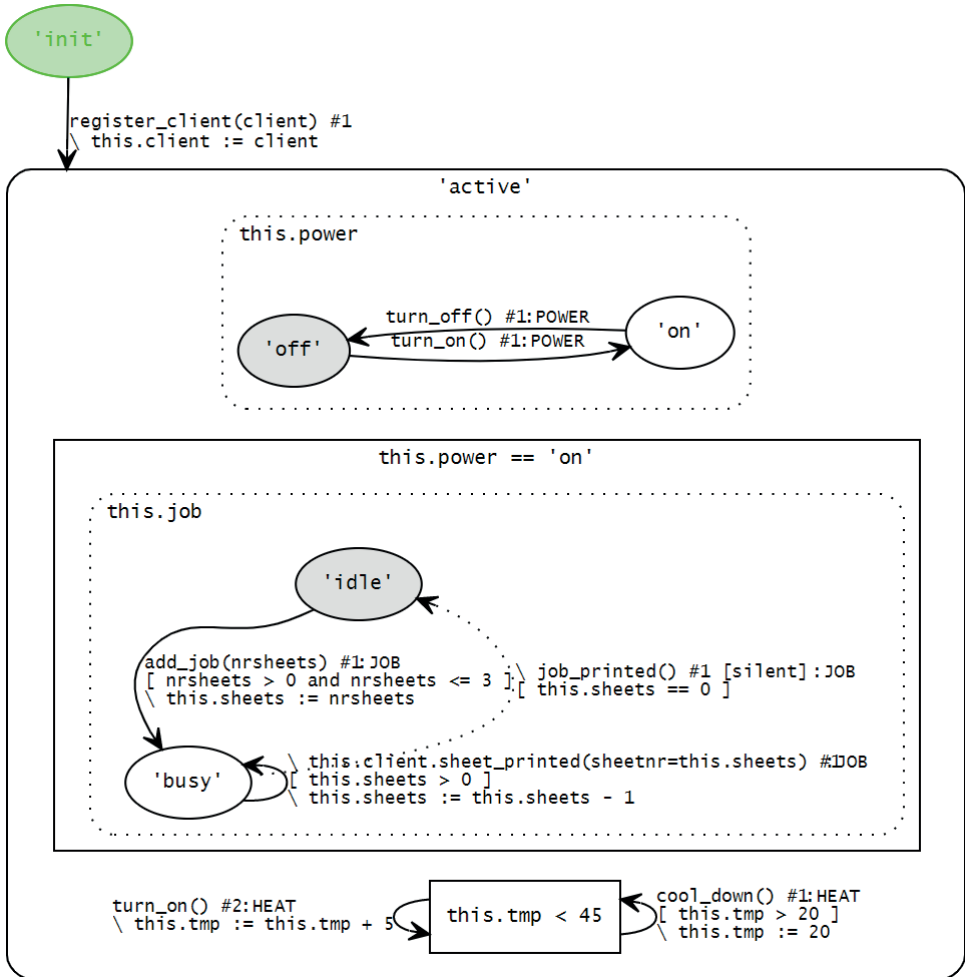


Figure 4.11: A visualisation of the example OIL specification of Fig. 4.10. States that are filled with a color correspond to the initial state.

```

1 module heat2ci
2 {
3     interface server
4     {
5         register_client(client: heat2ci.client)
6         turn_on()
7         turn_off()
8         add_job(nrsheets: int32)
9         cool_down()
10    }
11
12    interface client
13    {
14        sheet_printed(sheetnr: int32)
15    }
16 }

```

Figure 4.12: The IDL specification on which the OIL specification of Fig. 4.10 depends.

visualization of the running example.

Each OIL component specification defines a number of *instance variables* (lines 9-12), which store the state that the modeled component is in. Four types of instance variables are supported: boolean, enum, integer and component instance reference. Enum types can be defined within the specification itself (line 7).

There are three types of areas: *regions*, *states* and *zones* (lines 14-33). A region always refers to an enum variable and contains a number of states. These states each represent a value that the variable of its region can have. A zone has a boolean expression over variables and is used to restrict behavior.

The change of values for instance variables is triggered by the occurrence of *events*. Each event has an *operation*, which refers to the function being called. This operation may have parameters, which make it possible to transfer data between components. In the context of a component, the *cause* of an event can be either *reactive* or *proactive*. Reactive events are initiated by the environment, whereas proactive events are produced by the component itself, either sent to the environment or kept internally, the latter are also known as *silent* events.

Operations are declared in separate specifications in a language called IDL, short for Interface Definition Language (very similar to, but not to be confused with Microsoft's IDL⁴). Each IDL specification defines a number of *modules*, which contain *interfaces*, which in turn contain declarations of operations, possibly with parameters. A module may also contain enum type definitions, which can be used to define the type of a parameter. If one wants to refer to operations within an OIL component specification, the IDL modules in which they are defined must be imported (line 3, Fig. 4.10). Interfaces in the imported

⁴<https://learn.microsoft.com/en-us/windows/win32/midl/midl-start-page>


```

1  component heat2c
2  {
3    import heat2ci
4    provides heat2ci.server
5    requires heat2ci.client
6
7    enum power {off, on}
8    enum t_root_region {init, active}
9    enum t_job {idle, busy}
10
11   var power : power = 'off'
12   var client : heat2ci.client
13   var tmp : int32 = 20
14   var sheets : int32 = 0
15   var v_root_region : t_root_region = 'init'
16   var v_job : t_job = 'idle'
17
18   region root_region [this.v_root_region]
19   {
20     state init ['init']
21     state active ['active']
22     {
23       region power [this.power]
24       {
25         state off ['off']
26         state on ['on']
27       }
28
29       zone power_on [this.power == 'on']
30       {
31         region job [this.v_job]
32         {
33           state idle ['idle']
34           state busy ['busy']
35         }
36       }
37
38       zone heat [this.tmp < 45]
39     }
40   }
41   ...
42 }

```

Figure 4.13: The first half of the OIL specification of Fig. 4.10 after desugaring.

module can then be *provided* or *required* by the component (lines 4-5). The operation of a reactive event must be part of a provided interface and the operation of a proactive event must be part of a required interface.

The occurrence of an event corresponds to the firing of transitions labeled with that event (lines 35-57). Each transition has a source area (*in*), an event label (*on/do*), a target state (*go*) and a concern (*concern*), and optionally a guard (*if*), assignments (*assign*), an assertion (*assert*, not in example) and arguments for parameters (line 50, within parentheses).

4.3.3 Implementation Features

The desired implementation of OIL in Spoofox is required to realize a number of features. Though these features may not be complex to implement individually, the realization of the combination of these features can be. Below, we elaborate on each OIL feature (OF).

OF1: Multiple Syntaxes. OIL and IDL both offer an XML-based and a custom DSL syntax. Both syntaxes should be implemented; the XML syntax for backwards compatibility and because it is easier to parse for external tools, and the DSL syntax for a better user experience. It should be possible to transform a specification in one syntax into the other and all transformations to other targets should be available for both syntaxes.

OF2: Desugaring. OIL specifications allow some syntactic sugar, mainly in the form of leaving OIL concepts implicit, which reduces how much a user needs to write. For instance, in the running example, for region *job* (line 25) the variable reference is left implicit and for states *init*, *active*, *idle* and *busy* (lines 14, 15, 27 and 28) the values are left implicit, as well as corresponding enum types definitions. Also, the region for states *init* and *active* is left out. See Fig. 4.13 for how the first half of the running example would look like after desugaring. The implementation should be able to automatically desugar this and make the implicit information explicit.

OF3: Input Correctness. Not every specification is a correct OIL specification in terms of syntax or static semantics. Checking the static semantics of an OIL specification involves three types of analysis: structural checks, name resolution and type checking. The implementation should be able to check whether a specification meets all syntax and static semantics requirements and report useful errors to the user if it does not.

OF4: Language Interaction. IDL is a standalone language that can be used for other purposes than the context of OIL. OIL on the other hand should depend on IDL, both syntactically and semantically. Syntactically, because both languages use expressions and we want to minimize duplicate grammar definitions. Semantically, because module, interface, operation and parameter names in OIL specifications should refer to declarations in IDL specifications. The implementation should reflect this: IDL should be implemented separately and the implementation of OIL should depend on the implementation of IDL. This involves several forms of language composition [87] and language modularity [88, Sec. 4.6].

OF5: Multiple Targets. To represent the dynamic semantics of an OIL specification, it should be possible to translate OIL into other languages for which such semantics exists. For the formal verification of an OIL specification, the implementation should support a translation to mCRL2 [89]. For the execution of an OIL specification, the implementation should support a translation to GPL code.

4.4 CASE STUDY CONTEXT AND METHOD

In this section, we first describe the context of our case study. Next, we elaborate on our method for investigating the research question. Lastly, we describe the setup of our case study.

4.4.1 Context

Our evaluation focuses on two implementations of OIL, the Python implementation and the Spoofox implementation. The Python implementation was initially developed around 2011 by the third author and is still maintained by the third author to this day. The first author also worked on the Python implementation for a few months in 2016 as part of an internship within Canon Production Printing. The second author initiated the Spoofox implementation in 2018. A few months later, the first author also joined on the Spoofox implementation and both first and second author have been maintaining this implementation ever since. During this time, the third author was involved in the design decisions for the Spoofox implementation and some master students have contributed as well [90, 91, 92].

Before the Spoofox implementation was created, the second author was already familiar with language development in Spoofox. The second author has also been a contributor to Spoofox since before this study. The first author had limited experience in language development and no experience with Spoofox, but had some previous experience on rewriting and formal semantics. During the development of the Spoofox implementation, the developers had a close connection to the Spoofox development team for any questions and advice. All involved master students had no experience with Spoofox before they joined.

The third author is the creator of OIL, inspired by his prior research in the field of the specification of behavior [93]. The first author got experience with OIL during the internship, in which the goal was to understand and formalize the semantics of OIL by means of a(n) (initial) translation to mCRL2, on which the current translation to mCRL2 in Spoofox is based [94]. Prior to that, the first author had experience with behavior specification languages, specifically mCRL2. The second author had little experience with behavior specification languages before the development of OIL. All involved master students had no experience with OIL before they joined, but most had some experience with behavior specification languages.

4.4.2 Research Method

Productivity is about the amount of effort needed to implement some functionality. As proxy for effort we use code volume, as it is the only information available to us in this study that relates to effort. To represent functionality, we collect software artifacts relevant to language engineering that an implementation produces, such as parsers or transformations.

We compare the implementation of OIL in Spoofox with the implementation of OIL in Python. We do this by first gathering all artifacts relevant to language engineering. Then, for each artifact implemented in both implementations with similar functionality, we measure the code volume that is used to implement it and compare the measured code volume between the two implementations. In case parts of the code volume are reused for multiple artifacts or other projects, we measure it separately. Any dissimilarities between implementations of a language engineering artifact are discussed.

We use the *Source Lines of Code* [95] (SLOC) metric for measuring code volume, which excludes blank lines, comments, and lines only containing brackets from counting. In particular, we use the *Physical SLOC* metric [95], which considers each non-excluded line as a single line. This is in contrast to the *Logical SLOC* metric [95], which counts executable statements of which there could be multiple on a single physical line. Since the Spoofox meta-DSLs are declarative and the code written in these meta-DSLs do not necessarily correspond directly to statements or units of execution, we cannot measure Logical SLOC for both implementations. In the rest of this chapter, we use “SLOC” to refer to Physical SLOC.

We are aware that using code volume, quantified using a variant of the *Lines of Code* metric, is controversial and comes with advantages and disadvantages [96, 97, 95, 98]. However, an important motivation for using code volume per artifact as proxy for productivity is that it is an objective and repeatable measure and that it is applicable to both the Python and the Spoofox implementation.

Comparing code volume of the two implementations is only sensible when the volumes correspond to code that implements the same functionality. Since the two implementations do not always implement the exact same functionality, we first identify commonalities and differences before measuring volume. Then, in each implementation’s code volume measurement, we subtract lines for features or language constructs that are not in the other implementation to end up with a comparison of code that implements the same functionality. We do these measurements separately for artifacts in both implementations. We analyze where differences in code volumes originate from and to what extent parts of the implementations are reusable. We consider code to be reusable if it is generic enough such that it can be reused for other purposes, such as other language implementation, or for purposes outside of OIL altogether.

Depending on the artifact that is being compared, the relevant code consists of whole files or parts of files. When measuring code volume of whole files, we

use the `cloc` tool⁵ for the measurements, which counts SLOC and has builtin support for Python. To use this tool on measuring code written in Spoofox meta-DSLs, we manually add language definitions to `cloc` such that the tool can properly detect which lines need to be excluded from counting, such as lines with a single square bracket. When measuring code volume in parts of files, we count lines of code by hand. With the code measurements that we give, we also go into more detail on how we came to these measurements.

4.4.3 Setup

We answer the research question for the implementation aspects of language engineering separately. These are concrete syntax, abstract syntax, static semantics, dynamic semantics and design environment [3]. Since the design environment, which is about tool support for the language, is claimed to be (mostly) automatically derived by Spoofox, we do not consider this aspect separately, but as part of the other four aspects instead. Since the Python implementation does not have a dedicated text editor, this will only concern editor services such as syntax highlighting. Thus, we consider the following four aspects:

- **Concrete syntax** (Section 4.5): the textual representation of a language.
- **Abstract syntax** (Section 4.6): the internal representation of a language, including desugaring transformations defined on it.
- **Static semantics** (Section 4.7): the validity of specifications in a language.
- **Dynamic semantics** (Section 4.8): the execution of specifications in a language.

In each of these four sections we first highlight parts of the implementation that are relevant to the aspect. Afterwards, we evaluate Spoofox by answering the research question in the context of the aspect on a number of parts of the implementation, which we call *evaluation points*. For each evaluation point we structure the evaluation in the following parts:

- **Question:** what do we want to evaluate?
- **Method:** how are we going to evaluate this?
- **Results:** what is the information from the implementation(s) that is relevant for this evaluation point?
- **Analysis:** what does this information mean and how does it answer our question?
- **Conclusion:** what does the analysis give as answer to the question?
- **Discussion:** what else is relevant for this evaluation point?

We combine and summarize the findings of the evaluation points in Section 4.9.1. Since code volume per artifact does not exactly correspond to productivity [96, 97, 95, 98], our measurements are not directly representative for the research question. Therefore, in Section 4.9.2, we discuss the threats to

⁵<https://github.com/AlDanial/cloc>

the validity of our findings and how we have tried to counter them.

While our evaluation is mostly based on drawing conclusions from a quantitative analysis, we also make various observations on qualitative aspects. In Section 4.10 we discuss those observations on qualitative aspects. We discuss the strengths and weaknesses of Spoofox that we experienced and list the lessons we learned. We also propose an agenda of future work for Spoofox and discuss how some of the limitations that we encounter are already fixed in the next version of Spoofox.

4.5 CONCRETE SYNTAX

The Spoofox implementation of OIL comprises multiple syntactical (sub)languages for which the grammar is defined with SDF₃. It supports the original XML syntax of OIL and IDL as supported by the Python implementation, as well as a newly designed custom syntax, resulting in a total of four input languages and realizing OF₁ (Multiple Syntaxes). These input languages share common grammar rules for expressions, which touches on OF₄ (Language Interaction).

In this section, we describe the design of the existing and new syntaxes in Spoofox, their modular implementation, and the reuse of shared expression grammar rules and its implications on disambiguation. Also, we describe concrete syntax in the Python implementation and indicate how it differs from the Spoofox implementation. These descriptions form the sources of information for the evaluation that follows, where we answer the research question on productivity for the concrete syntax aspect of OIL's implementation in Spoofox.

4.5.1 From XML to Custom Syntax

Implementing a language with concrete syntax in Spoofox requires a grammar written in SDF₃. The grammar specific for the OILXML subset of XML consists of a grammar rule for each XML element, with, if applicable, a list of specific attributes of the element and, if applicable, a list of child elements. Fig. 4.14 shows an excerpt of the SDF₃ grammar of OILXML for the transition concept (see Section 4.3.2). Fig. 4.16a contains an example transition in OILXML and Fig. 4.16c contains the corresponding abstract syntax, expressed in ATerm (see Section 4.2.2).

The original Python implementation uses XML as its syntax because of two reasons. First, other projects at Canon Production Printing already used XML and thus engineers are familiar with it. Second, for XML an off-the-shelf parser could easily be used. However, writing XML specifications by hand is not user friendly [2, p. 101]. Although a custom syntax was desired already in the Python implementation to improve usability, implementing it was considered too much work. With Spoofox's language-oriented programming view [99], re-implementing OIL in Spoofox made it much more feasible to design a custom syntax for OIL, dubbed "OILDSL".

The most prominent problem of XML syntax is the syntactic noise of XML elements and attributes. Still, the same high-level structure of OIL's concepts from OILXML was used as a basis for designing the new syntax. By doing

```

1 context-free syntax
2
3 Transition.XMLTransitionSimple = [
4   <transition[{{TransitionAttr ""}}+]/>
5 ]
6
7 Transition.XMLTransitionComplex = [
8   <transition[{{TransitionAttr ""}}*]>
9     [TransitionSelf?]
10    [{{TransitionParameter "\n"}}*]
11    [TransitionReturn?]
12    [TransitionGuard?]
13    [TransitionAssignments?]
14    [TransitionAssert?]
15   </transition>
16 ]
17
18 TransitionAttr.XMLMessageCauseAttr = [cause="[Cause]"]
19 TransitionAttr.XMLMessageOperationAttr = [operation="[ID]"]
20 TransitionAttr.XMLSourceAttr = [source="[AreaReference]"]
21 TransitionAttr.XMLTargetAttr = [target="[AreaReference]"]

```

Figure 4.14: An SDF3 code snippet of the grammar of transitions in OILXML.

```

1 context-free syntax // Transitions
2
3 Transition.DSLTransition =
4   [{{TransitionElement " "}}+ end]
5
6 TransitionElement.DSLMessage =
7   [[MessageCause] [MessageEventType?] [MessageOperation?]]
8
9 MessageCause.DSLMessageCause = Cause
10 Cause.DSLReactive = [on]
11 Cause.DSLProactive = [do]
12
13 TransitionElement.DSLSource =
14   [in [AreaReference]]
15 TransitionElement.DSLTarget =
16   [go [AreaReference]]

```

Figure 4.15: An SDF3 code snippet of the grammar of transitions in OILDSL.

```
<transition cause="reactive" operation="turn_on"
  source="off" target="on"/>
```

(a) Example transition in OILXML.

```
in off on turn_on() go on end
```

(b) The transition of (a) translated to OILDSL.

```
XMLTransitionSimple(  
  [ XMLMessageCauseAttr(XMLReactive())  
    , XMLMessageOperationAttr("turn_on")  
    , XMLSourceAttr(AreaReference(["off"]))  
    , XMLTargetAttr(AreaReference(["on"]))  
  ]  
)
```

(c) The OILXML AST that corresponds to (a).

```
DSLTransition(  
  [ DSLSource(AreaReference(["off"]))  
    , DSLMessage(  
      DSLMessageCause(DSLReactive())  
      , None()  
      , Some(DSLMessageOperation("turn_on", [], None()))  
    )  
    , DSLTarget(AreaReference(["on"]))  
  ]  
)
```

(d) The OILDSL AST that corresponds to (b).

Figure 4.16: An example transition in OILXML and OILDSL with the corresponding ASTs.

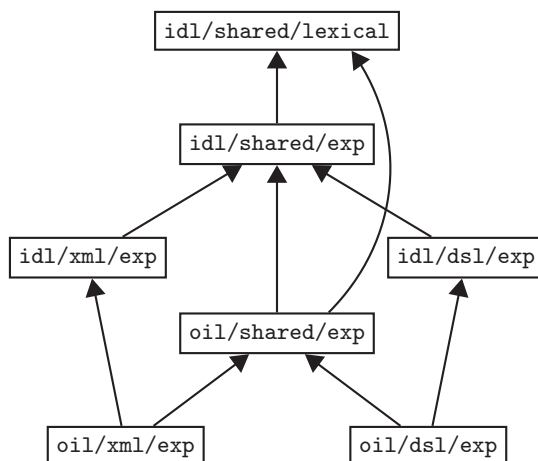


Figure 4.17: A (simplified) import graph of IDL’s and OIL’s expression grammars, that depicts how modules are reused. Node labels correspond the SDF3 module names. Arrows mean “imports”.

so, the abstract syntax of both syntaxes are similar, which eases forward and backward migration between both syntaxes. Without the noise of XML elements in OILDSL, the syntax for transitions becomes simpler; XML open and closing elements are replaced with simple keywords and brackets. Fig. 4.15 shows an excerpt of the SDF3 grammar for transitions in OILDSL. Fig. 4.16b shows the concrete syntax in OILDSL that corresponds to the OILXML variant in Fig. 4.16a, and Fig. 4.16d depicts the corresponding AST.

4.5.2 Composed Grammars and Disambiguation

The four input languages (IDLXML, IDLDSL, OILXML and OILDSL) share parts of their grammars, especially in the context of expressions: the expression grammar of OIL extends the expression grammar of IDL, and the XML and DSL expression grammars only differ in a few operators. To prevent the definition of duplicate grammar rules across the SDF3 grammar definitions of the input languages, the grammar definitions are split up into several reusable modules.

See Fig. 4.17 for all SDF3 modules that define grammar rules for expressions. Modules `idl/xml/exp`, `idl/dsl/exp`, `oil/xml/exp` and `oil/dsl/exp` define the expression grammar for IDLXML, IDLDSL, OILXML and OILDSL respectively. Shared grammar is defined in separate modules and reused for multiple grammars by means of `import` statements. As an example, Fig. 4.18 shows the composition of the IDLXML expression grammar with the shared OIL expression grammar (module `oil/shared/exp`) to obtain the OILDSL expression grammar in SDF3. The XML specific grammar rules are defined once for IDLXML in module `idl/xml/exp` and are reused for OILXML by importing them in module `oil/xml/exp`. Also, the OIL-specific grammar rules, used

```

1  module idl/xml/exp
2
3  imports idl/shared/exp
4
5  context-free syntax
6  Exp.XMLLt  = [[Exp] &lt; [Exp]] {left}
7  Exp.XMLLeq = [[Exp] &lt;= [Exp]] {left}
8  Exp.XMLGt  = [[Exp] &gt; [Exp]] {left}
9  Exp.XMLGeq = [[Exp] &gt;= [Exp]] {left}
10
11 context-free priorities
12 { left: Exp.Plus Exp.Minus } >
13 { left: Exp.XMLLt Exp.XMLLeq Exp.XMLGt Exp.XMLGeq } >
14 { left: Exp.Eq Exp.Neq }

```

```

1  module oil/shared/exp
2
3  imports idl/shared/Lexical
4  imports idl/shared/exp
5
6  context-free syntax
7  // OIL-specific expressions
8  Exp.Reference = ID
9  Exp.Old      = [[Exp]']
10
11 context-free priorities
12 Exp.Has >
13 Exp.Old >
14 { left: Exp.Not Exp.Length }

```

```

1  module oil/xml/exp
2
3  imports idl/xml/exp
4  imports oil/shared/exp

```

Figure 4.18: SDF₃ modules that define the expressions syntax for OILXML (simplified). The `Exp.Reference` constructor defines a variable reference using the lexical ID sort. The `Exp.Old` constructor defines a suffix operator for referencing old values of a variable, which can be used in assertions.

for both OILDSL and OILXML, are imported from module `oil/shared/exp`. Together, both imported modules form the expression grammar of OILXML.

Module `idl/xml/exp` has emerging ambiguities between the XML-specific operators and other operators. Syntactical ambiguity is when a single input program can be parsed to multiple different trees using the same grammar. For example, `1 + 2 >= 3` could be parsed in two ways: `(1 + 2) >= 3` and `1 + (2 >= 3)`. With priorities in SDF₃, one can define the relative precedence of expression operators. Since priorities are transitive, only priorities with respect to direct neighbors in the priority order (operators `Plus/Minus` and `Eq/Neq`) are required for the new operators. Because `Plus` gets a higher priority than `XMLGeq`, the previous example will be parsed as `(1 + 2) >= 3`. In module `oil/xml/exp`, no additional disambiguation is required, as no more new combinations of expression operators arise and no other syntactical ambiguities emerge.

4.5.3 *The Python Implementation*

The Python implementation only supports IDLXML and OILXML. The parser for these languages consists of two stages. First, the XML structure is parsed using the `Minidom` library⁶. Second, the expressions inside XML elements are parsed using the `Pyparsing` library⁷. For both stages, the implementation uses a custom layer on top of the libraries to support grammar specification and parser implementation. For example, the Python implementation uses a metamodel expressed in a custom XML subset that defines the restrictions of OILXML with respect to generic XML.

Fig. 4.19 shows an excerpt of this metamodel. Lines 2–4 define a regular expression for identifiers that can be referred to in other parts (see, e.g., line 10). Lines 6–33 define expressions, in which the order of levels of operators indicate the precedence between operators. Lines 35–47 define the transition concept. A Python script parses this metamodel and checks for an input OIL program, parsed using `Minidom`, whether it conforms to the metamodel. Helper functions on top of `Pyparsing` ease the definition of expression grammar rules, e.g., by automatically allowing whitespace around operators. The levels of operators are used to define precedence in `Pyparsing`. The custom layer is not specific to OIL and can be reused for other XML languages with embedded expressions.

In Section 4.3.1 we have described the Python implementation’s environment for viewing and editing OIL specifications. Several editor services related to viewing and editing concrete syntax similar to those in regular IDEs are available, which we detail below. For viewing OIL specifications, the Python implementation supports pretty printing and origin tracking (for error reporting; see Section 4.2.3), which are implemented manually. Both pretty printing and origin tracking are implemented in Python based on the data structures that result from the parsers.

For editing OIL specifications, the Python implementation has limited cus-

⁶<https://docs.python.org/3/library/xml.dom.minidom.html>

⁷<https://pypi.org/project/pyparsing/>

```

<metamodel name="oil">
  <regexp name="identifier" pattern="[_A-Za-z][_0-9A-Za-z]*">
    <documentation>A standard programming identifier.</documentation>
  </regexp>
  ...
  <parexp name="expression">
    <level>
      <operator symbol="number" pattern="0|[1-9][0-9]*"/>
      <operator symbol="new" pattern="new {qualified_identifier}"/>
      <operator symbol="identifier" pattern="{identifier}"/>
      ...
    </level>
    ...
    <level>
      <operator symbol="_+ "/>
      <operator symbol="_ - "/>
    </level>
    <level>
      <operator symbol="_ = "/>
      <operator symbol="_ ! = "/>
      <operator symbol="_ < ; = "/>
      <operator symbol="_ > ; = "/>
      <operator symbol="_ < ; _ "/>
      <operator symbol="_ > ; _ "/>
    </level>
    <level>
      <operator symbol="_ and _ "/>
    </level>
    <level>
      <operator symbol="_ or _ "/>
    </level>
    ...
  </parexp>
  ...
  <entity name="transition">
    <documentation>A transition specifies a rule that indicates for a certain
      action when and how variables change.</documentation>
    <generalization entity="actionable"/>
    <generalization entity="compositional"/>
    <generalization entity="concernable"/>
    <child entity="self" lower="0" upper="1"/>
    <child entity="argument" lower="0" upper="inf"/>
    <child entity="result" lower="0" upper="1"/>
    <child entity="guard" lower="0" upper="1"/>
    <child entity="assignment" lower="0" upper="inf"/>
    <child entity="assertion" lower="0" upper="1"/>
  </entity>
  ...
</metamodel>

```

Figure 4.19: An excerpt of the Python implementation's metamodel for OIL, expressed in a custom chosen subset of XML. OIL specifications are checked to conform with this metamodel using a handwritten but generic Python script.

tom support. The web-based Python implementation does not include an editor, so external tools are used instead for editing OIL specifications, typically the Visual Studio IDE. To support the editing of OIL specifications, an XSD (XML Schema Definition Language⁸) file is generated from the metamodel using a handwritten script. XSD schemas define a subset of the XML language. A generic IDE plugin for XML uses this XSD file to provide syntactic code completion and error recovery. Although XSD could also be used for realizing a parser, it was only used for realizing limited editor support because XSD was found not to be expressive enough to cover all syntactical aspects of OIL.

4.5.4 Evaluation

To evaluate the productivity of implementing concrete syntax, we look at a single evaluation point: the complete implementation of concrete syntax in OIL's implementations in Spoofox and in Python. We consider seven concrete syntax artifacts: the grammar, the parser, and the editor services pretty printing, origin tracking, syntax highlighting, error recovery and syntactic code completion.

Question. Does it cost less code volume to implement the artifacts for OIL's concrete syntax in Spoofox compared to Python?

Method. First we identify for each of the seven artifacts related to concrete syntax to what extent they are realized in the SDF₃ and Python implementations of OIL. Then, per artifact, we measure and compare the code volume (in terms of SLOC [95]) related to the artifact in each implementation.

For the sake of comparability, we want to compare the lines of code of the implementations where they implement the exact same syntax. The Python implementation only contains OILXML and not OILDSL. Thus, in the Spoofox implementation we consider the grammar except those parts specific to OILDSL, i.e., we consider the OILXML-specific parts and the parts shared between OILXML and OILDSL. Since the syntactic languages of both implementations are not exactly the same, we subtract lines from our measurements that concern syntactical elements not present in the other implementation. In the Python implementation's metamodel, we manually exclude tags that are specific for documentation from the counting, as we consider them as comments in the definition of SLOC. Since XML and expression parsing are separately implemented in Python, we measure and analyze those separately.

Both the Python as well as the Spoofox implementation of OIL could be seen as consisting of OIL-specific code and more generic, reusable code. We consider code to be reusable if it is generic enough such that it can be reused in the implementation of a language other than OIL. In the Spoofox implementation, we reuse code from the standard library of Spoofox, and we do not include it or the implementation of the language workbench itself in the measurements. In the Python implementation, both OIL-specific and reusable code are implemented manually, which we therefore both measure. We count OIL-specific code separately from reusable code. For the reusable code, we analyze to what extent it can be reused.

⁸<https://www.w3.org/XML/Schema>

Syntax impl. artifact	Spoofox	Python
Grammar (excl. expressions)*	● 274	● 161
Grammar (expressions)*	● 91	● 41
Parser generator (excl. expressions)	● 0	● 344
Parser generator (expressions)	● 0	● 298
Pretty printing	● 0	◐ 90
Origin tracking	● 0	● 205
Syntax highlighting	● 0	◐ 0
Error recovery		
Syntactic code completion	● 0	◐ 121
Total (OIL-specific)*	365	202
Total (All)	365	1260

Table 4.1: Code volume (in SLOC) for the Spoofox and Python implementations of OILXML’s concrete syntax, counted per implementation artifact. * indicates OIL-specific artifacts. ● = implemented; ◐ = partially implemented (only for XML, not for expressions).

Results. Table 4.1 gives an overview of which syntax artifacts are available in each implementation and states the SLOC per artifact. The Spoofox syntax implementation of OILXML comprises 365 SLOC of SDF₃ grammar and the Python implementation comprises 1260 SLOC. Spoofox realizes all artifacts in full from this grammar: a parser, a pretty printer, origin tracking, and editor services such as syntax highlighting, error recovery, and code completion. The Python implementation contains a parser and origin tracking for the full language. Other artifacts are only implemented for XML and not for the expressions inside XML: pretty printing, syntax highlighting, error recovery, and code completion. To ensure that our comparison is on two implementations that cover the same syntactical language, in the Spoofox source we have withheld the syntactical elements that are not in Python from counting (31 out of 396 SLOC deducted from original source; 7.8%) and in the Python source we have withheld elements that are not in the Spoofox implementation from counting (46 out of 1306 SLOC deducted from original source; 3.5%). Of the Python implementation, only the grammar (202 out of 1260 SLOC) is specific to OIL, which relates to the metamodel.

Analysis. We analyze our results by first comparing the total code volumes of both implementations and then compare per syntax implementation artifact.

The results show that the syntax implementation artifacts of OIL are realized in the Spoofox implementation with a factor of 0.29 SLOC compared to the Python implementation. All 365 Spoofox SLOC are OIL-specific. In the Python implementation, only 202 SLOC is specific to OIL, namely for defining the metamodel; the rest is reusable for XML-based languages with embedded expressions. The Spoofox implementation realizes all syntax artifacts for the full language, whereas the Python implementation only realizes the parser

and origin tracking for the full language; the other artifacts produced by the Python implementation do not provide support for expressions. The Spoofox implementation consists of SDF₃ only, i.e., the grammar, from which all other six artifacts are derived. In the Python implementation, most code is attributed to realizing the parser. For the other artifacts, additional code was needed ($90 + 205 + 121 = 416$ SLOC). In the Python implementation, syntax highlighting for XML was the only editor service that was available without manual implementation by using an existing XML plugin in Visual Studio.

The Python implementation uses a custom layer on top of existing libraries for XML parsing (Minidom) and expressions parsing (Pyparsing). First, the Python implementation uses a metamodel (202 SLOC) and a script that checks whether OIL models conform to the metamodel (344 SLOC). Second, the Python implementation adds helpers on top of Pyparsing that prevent repeating low-level grammar patterns (298 SLOC). Although the custom layer for expressions is reusable, it is more restrictive than SDF₃ as, e.g., it only supports left associativity. The SDF₃ implementation did not use code in addition to the grammar, which is the main reason why the Spoofox implementation contains fewer SLOC.

In Spoofox, the OILXML grammar is defined with a total of 21 SDF₃ modules. Out of the 365 SLOC used to define these modules, about 28% exists purely to compose these modules. This consists almost only of module name definitions and import statements. For some grammar modules, such as those that define the expression grammar, the split up into smaller modules is beneficial, because it enables reuse of grammar rules for the other input languages as described in Section 4.5.2. A third of the grammar modules however are only used once. These modules could have been merged with the modules that use them instead, which would have saved a few SLOC. We see this difference as negligible, as this only saves on import statements, which do not directly define the grammar of the language. In Python, the grammar is defined in a single metamodel, so no SLOC is used for composition.

Pretty printing is implemented manually in the Python implementation (90 SLOC). This is a generic XML pretty printer, not specific to OILXML. For expressions, it simply copies the textual representations of expressions to output programs. In the Spoofox implementation, pretty printing is automatically derived based on the formatting of the grammar rules in SDF₃. For example, lines 7–16 of Figure 4.14 define the pretty printing of transitions to be spread over multiple lines and with indented child elements. These templates increase the number of SLOC used for defining the grammar rule, as without formatting the complete rule could be at a single line.

Origin tracking (see Section 4.2.3), is generated automatically from the SDF₃ grammar. In the Python implementation, origin tracking is only provided without requiring additional implementation by the Pyparsing library used for parsing expressions. For parsing XML, a manual implementation was needed to realize origin tracking (196 SLOC). The origins returned by the Pyparsing library used for expressions are relative. Absolute source locations for expressions are calculated by adding the relative offsets of expressions to

the parent XML tag content's source location (9 SLOC, reported under "Origin tracking" in Table 4.1).

Syntax highlighting is the only editor service in Python that is realized without additional effort, using Visual Studio's default XML plugin. Note, however, that this only supports syntax highlighting for the XML part of OIL. For the expressions inside XML tags, it does not support syntax highlighting. The Spoofox implementation does support syntax highlighting for the complete language.

The Python implementation realizes error recovery and code completion for the XML part of the language by generating an XSD schema from the metamodel. The script that generates the XSD schema (121 SLOC) is reusable, not specific to OIL. Given the XSD schema, the editor's default XML plugin provides error reporting on invalid XML tags and it provides code completion. The Spoofox implementation supports error recovery and code completion for the complete language without additional implementation, by automatically generating the editor services based on the SDF₃ grammar.

Conclusion. The Python implementation uses less OIL-specific code to implement OIL's concrete syntax (metamodel of 202 SLOC) than Spoofox (SDF₃ grammar of 365 SLOC). However, whereas Spoofox does not require code in addition to the grammar to fully implement a range of editor services, the Python implementation requires additional code for implementing a parser and other editor services. This is in spite of the Python implementation making use of existing libraries for XML parsing and expression parsing and the Python implementation realizing some editor services not for the full OIL language. The code in the Python implementation, apart from the metamodel, is reusable in the sense that it is not specific to OIL. The reusable parts of the Python implementation can be reused for implementing other languages with XML syntax with a restricted form of embedded expressions.

Comparing the implementations, including the reusable parts, the Spoofox implementation realizes all editor services for the full OIL language using less than one third of SLOC. Therefore, the results show that it costs less code volume to implement the artifacts for OIL's concrete syntax compared to Python. This is especially the case when, in addition to only a parser, editor services are required for interactive use in IDEs.

Discussion. The Python implementation uses, on top of the existing parsing libraries, a custom layer to support the implementation of OILXML's syntax. This custom layer enables the use of the metamodel and prevents repeating low-level grammar patterns. Alternatively, OILXML's syntax could also have been implemented directly using the existing libraries. On one hand, the custom layer is reusable and costs extra code. On the other hand, the custom layer saves code by preventing the need to repeat low-level implementation patterns. Although we cannot make a comparison with a Python implementation that does not include the custom layer, we expect that such an implementation could be smaller than the current Python implementation. An implementation without reusable parts could be more specific to OIL and therefore possibly smaller. In the Spoofox implementation, SDF₃ was sufficient to implement

OILXML and OILDSL.

Implementing the parser for OILXML in Python takes about twice the SLOC as using SDF₃. When editor support is not relevant, one could argue whether the factor two fewer SLOC is reason enough to start using a language workbench. However, we expect that the factor is relatively low because XML syntax is used. By using XML, the existing Minidom library could be used, but this also imposes the restriction of only supporting subsets of XML. For custom syntax, e.g., for OILDSL, we expect that a Python implementation using Pyparsing would cost relatively more SLOC than in SDF₃, as a complete grammar needs to be implemented in more detail. The use of another parser generator library in Python could bring this closer to SDF₃. In Spoofox, the code volume specific to the OILDSL grammar is actually smaller than the code volume specific to the OILXML grammar due to OILDSL being more concise: the Spoofox implementation contains 219 OILXML specific SDF₃ SLOC and 168 OILDSL specific SDF₃ SLOC.

The grammar of OILXML imposes a few restrictions on the order of attributes or child elements of an XML element. This was originally done so that the structure of the derived AST can be made slightly simpler, resulting in slightly simpler transformations from this AST. To lift these restrictions, we believe that about 10 SDF₃ SLOC would be necessary. The Python implementation does not have such restrictions.

Part of the Python implementation's editor services, e.g., origin tracking, are implemented only for the web-based tooling which only statically displays specifications and errors and does not support editing specifications. Therefore, the editor services are only used in a static way, in contrast to the interactive way in IDEs. We expect that extending the editor services in the Python implementation to have support for interactive use would cost more code.

4.6 ABSTRACT SYNTAX

The abstract syntax of a language defines how a language is represented internally. For textual languages, such as OIL, this is done by means of AST schemas. Given an SDF₃ grammar definition, a corresponding AST schema is generated automatically. To structure the Spoofox implementation of OIL, additional intermediate representations have been defined, which we discuss here. We also describe the transformation architecture that is shaped around these intermediate representations, specifically focusing on desugaring transformations and on the resilient staging framework that serves as the basis of this architecture, which realize OF₂ (Desugaring). Afterwards, we discuss how OIL is internally represented in the Python implementation. Lastly, we evaluate Spoofox on productivity in the context of abstract syntax.

4.6.1 *Intermediate Representations*

In addition to the AST schemas that are automatically generated by SDF₃ for OILDSL and OILXML, three intermediate representations (IRs) are defined for OIL:

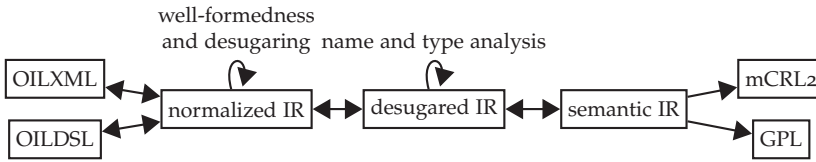


Figure 4.20: An overview of the transformation architecture of the implementation of OIL in Spoofax. Boxes correspond to AST schemas and arrows correspond to transformations.

- **Normalized IR** A representation that acts as a middle ground between OILDSL and OILXML while still containing as many syntactic details from both languages as possible.
- **Desugared IR** A simplified representation where syntactical details are removed and implicit details are made explicit to enable concise specification of static semantics.
- **Semantic IR** A representation that restructures an OIL specification to ease the implementation of dynamic semantics (code generation) of OIL.

The use of IRs provides separation of concerns: each IR is related to a different language implementation aspect. See Fig. 4.20 for how the IRs fit in the transformation architecture. Transformations are defined between the normalized IR and both the OILXML and OILDSL AST schemas in both directions so that one can easily switch between OILXML and OILDSL [36]. Any transformations that follow are independent of the concrete syntax used.

To illustrate some differences between IRs, we use the same transition as the example in Fig. 4.16. See Fig. 4.21 for this transition in the normalized, desugared and semantic IR. One notable change from OILDSL and OILXML to the normalized IR is that the transition term in the normalized IR now has fixed subterms instead of a list of terms, which was done to make it easier to define transformations on it. When moving to the desugared IR, some optionality is removed by removing `Some` wrapper terms and by replacing `None` terms with information made explicit, such as `Call`. When moving to the semantic IR, transitions are now grouped per event. This is useful for code generation, as an OIL specification is executed by means of sending or receiving events. Also, the source and target are used to define the transition pre- and postconditions (with `ConditionReference`) and the transition update (with `UpdateReference`).

4.6.2 Desugaring

Before the normalized IR can be transformed to the desugared IR, a number of desugaring transformations are applied first. There are a total of 14 desugaring transformations defined which all use the normalized IR both as input and output. Most of the desugaring transformations are *explicitations*, which make implicit information explicit. These are necessary to be able to remove the

```

Transition(
  Some(
    Message(
      Some(MessageCause(Reactive()))
      , None()
      , Some(MessageMethod("turn_on", []))
    )
  )
, Some(Source(AreaReference(["off"])))
, Some(Target(AreaReference(["on"])))
, ...
)

```

```

DESTransition(
  DESMessage(
    Some(Reactive())
    , Call()
    , DESMessageMethod("turn_on", [])
  )
, "off"
, "on"
, ...
)

```

```

SEMEvent(
  "heat2ci_server_turn_on_call_reactive"
, Reactive()
, Call()
, "heat2ci"
, "server"
, SEMMethod("turn_on", [])
, [ SEMTransition(
    ...
    , ConditionReference("off")
    , [UpdateReference("on")]
    , ConditionReference("on")
  )
, ...
]
)

```

Figure 4.21: The transition of Fig. 4.16 in the normalised, desugared and semantic IR respectively.

```

1  oil-auto-value = topdown(try(oil-auto-value-term))
2
3  oil-auto-value-term:
4    State(name, None(), supers, areas) ->
5    State(name, <oil-auto-value-new> name, supers, areas)
6
7  oil-auto-value-new = newname ; !Some(StateValue(EnumLiteral(<id>)))

```

Figure 4.22: The Stratego implementation of the *auto-value* desugaring transformation.

optionality of terms when transforming to the desugared IR.

See Fig. 4.22 for the implementation of one of the (simpler) explication transformations defined in the Stratego implementation. This transformation, called *auto-value*, gives every state a value if it does not (explicitly) have one. The rule defined on line 1 traverses top-down over the AST to try and apply the rule *oil-auto-value-term* on every node. This rule, defined on lines 3-5, does the actual explication: if a state without a value is found (line 4), the value is added (line 5). It uses the rule defined on line 7, which creates a fresh name for the state value given the name of the state.

4.6.3 Resilient Staging

To keep the desugaring transformations simple, they each have expectations on the input. For instance, desugaring transformation *auto-type* that derives new types from the areas of an OIL specification expects that each state has a value. Most of these expectations are ensured by other desugaring transformations. For instance, *auto-value* ensures that every state has a value, which matches the expectation of *auto-type*.

To help us structure the desugaring transformations, as well as the transformation architecture as a whole, a framework that we call *resilient staging* is used. This framework is based on *stages*, which are equipped with a precondition, a transformation, and a postcondition. Each stage should only have one specific transformation purpose to keep them well maintainable and reusable. Stages can be concatenated to create larger transformations, which we call *pipelines*.

The precondition represents requirements on the input of the stage, such as the presence or absence of specific terms or term patterns. When executing a stage, the precondition is checked first. If the precondition is met, the actual transformation will be executed. Otherwise, the pipeline stops and reports the errors from the precondition. For some stages, checking the precondition may require work that is useful for the transformation itself too, such as collecting specific terms. To prevent duplicate work, the precondition may also pass data to the transformation if the precondition is met.

After the transformation has been executed, the postcondition is checked. This postcondition represents requirements on the output of the stage, effectively testing whether the transformation was successful. If the postcondition is

```

stage(pre, trans, post):
  StageSuccess(ast) -> result
  where
    //check preconditions
    (pc-data, errors) := <pre> ast;
    if (<?[]> errors) then

      //do the transformation
      ast' := <trans> (pc-data, ast);

      //check postconditions
      errors' := <post> ast';
      if (<?[]> errors') then
        result := StageSuccess(ast')
      else
        result := StageFailure(ast', errors')
      end
    end

  else
    result := StageFailure(ast, errors)
  end
end

```

Figure 4.23: The Stratego transformation rule to define a stage (simplified).

not met, the pipeline is aborted and errors are returned. Ideally, postconditions checking should only be enabled during development, since stages should be correct when used in production.

In a sense, the pre- and postcondition provide a contract over the transformation: they define what is required by the transformation and what can be expected from the transformation. A clear contract and transformation purpose indicate how and where transformations should be embedded into pipelines, also promoting reusability. When a transformation is used or defined incorrectly, the stage conditions will show what and where the issue is, hence “resilient” in resilient staging.

In Stratego, stages are defined using the transformation rule `stage` shown in Fig. 4.23. The three parameters `pre`, `trans` and `post` are the precondition, transformation and postcondition respectively. The precondition and postcondition are transformations too, which return a list of errors, given an input AST. More on this in Section 4.7.1. Whenever a condition returns errors, the pipeline is abandoned and the errors are returned. See Fig. 4.24 for the instantiation of the stage of *auto-value*.

4.6.4 The Python Implementation

In the Python implementation no intermediate representations are used. The representation that results from parsing OILXML is used directly for checks and transformations. See Fig. 4.25 for the general transformation architecture.

```

1 oil-auto-value-stage =
2   stage(
3     stage-preconditions-true,
4     oil-auto-value,
5     all-states-value
6   )

```

Figure 4.24: The creation of the stage for *auto-value* in Stratego, using a generic rule *stage-preconditions-true*, the transformation from Fig. 4.22 and a postcondition rule *all-states-value*.

well-formedness, desugaring,
name and type analysis



Figure 4.25: An overview of the transformation architecture of the implementation of OIL in Python. The rectangular box is an AST schema, the wavy box is text, and the arrows are transformations.

This representation consists of two parts: the AST representation generated by the Minidom parser and the AST representation generated by the expression parser. For the Minidom AST representation, many helper functions have been defined that hide the use of Minidom, mainly for the access or derivation of information from it. Many of such information is cached so that the Minidom AST does not need to be accessed too frequently. For the expression AST representation, a custom *Expression* class is defined that is used to represent any expression term.

Desugaring transformations are defined as Python functions that traverse the AST and apply the changes where necessary. See Fig. 4.26 for the implementation of *auto-value* in Python. It traverses the AST to find all states (line 3). Then if this state does not have a value (line 4), a new value name is created based on the name of the state (lines 5 and 7), a new AST element is created that holds this value (lines 6 and 8), which is then added to the state (line 9).

The Python implementation does not explicitly define pre- and postconditions per transformation like resilient staging does, but it does define transformations and conditions on the AST as separate functions, which are called in a specific (interleaving) order.

4.6.5 Evaluation

To evaluate the productivity of implementing abstract syntax, we look at two evaluation points: AST representations and desugaring transformations.

AST representations. **Question.** Does it cost less code volume to define AST representations for OIL in Spoofox compared to Python?

```

def addAutoStateValues(spec):
    dom = spec.getDom()
    for state in spec.getElements('state'):
        if len(Children(state, ['value'])) == 0:
            stateName = state.getAttribute('name')
            autoValue = dom.createElement('value')
            literalName = spec.createUniqueLiteralName(stateName)
            _setExpressionText(spec, autoValue, f'"{literalName}"')
            state.insertBefore(autoValue, state.firstChild)
            state.hasAutoTerm = True
    spec.initElementCache()

```

Figure 4.26: The Python implementation of the *auto-value* desugaring transformation.

Method. We collect all AST representations that are used in the transformation architecture of an OIL specification in both the Spoofox and the Python implementation and measure the SLOC used to define them.

Results. As was shown in Fig. 4.20, the Spoofox implementation defines seven AST schemas for OIL: OILXML AST, OILDSL AST, the three IRs, mCRL2 AST and GPL AST. As was shown in Fig. 4.25, the Python implementation defines one AST representation. This AST representation is split into two parts: a Minidom AST representation and an expression AST representation.

In the Spoofox implementation, all AST schemas, apart from the normalized and desugared IR, are automatically derived from their grammar defined in SDF₃. The normalized and desugared IR do not have their own grammar and have their constructors defined in Stratego instead. Their AST schemas are defined with 43 SLOC and 23 SLOC respectively, with a shared expression AST schema defined with 33 Stratego SLOC. In the Python implementation, the constructors for the Minidom AST representation are automatically derived from the Minidom parser. The Expression class used for the expression AST representation is defined with 9 Python SLOC.

Analysis. If a grammar already exists, the Spoofox implementation does not need any SLOC to define the AST schemas as they are generated automatically. This is also the case in the Python implementation for the Minidom AST representation. If the grammar is not available, an AST schema can be defined in Spoofox with 1 Stratego SLOC per constructor, as is the case for the normalized and desugared IRs. In Python, the expression AST representation only consists of one constructor, namely a general Expression constructor with 7 children, defined with 9 SLOC. Due to this constructor, the expression AST representation in Python is more general than the expression AST schema in Spoofox, as the latter explicitly defines a constructor for each type of expression. This is also the reason why the expression AST schema in Spoofox uses more SLOC than the expression AST representation in the Python implementation.

Conclusion. Due to the differences in available AST representations, we cannot give a conclusion on the definition of AST representations. For the

AST representation that is available in both implementations, namely that for expressions, we cannot draw a conclusion either, due to the different approach for constructors and the small size.

Discussion. The difference between the Spoofox and Python implementations in the use of IRs is partially due to the mutability of ASTs. ATerm terms are immutable, so any desired change to the definition of an AST requires the definition of new constructors. In the Python implementation, the AST is mutable, so the AST can be changed dynamically. This does have the downside that it is more difficult to know exactly what information is available at any point in the transformation architecture. Due to the mutability of the AST in the Python implementation, it was found during its development that the definition of explicit IRs would not be worth the effort for the benefits it could bring over using the single mutable AST.

An interesting observation is the difference in the approach of defining the expression AST schema between Spoofox and Python. In Spoofox each type of expression is defined with an explicit constructor, while the Python implementation uses one generic constructor. This is related to the transformation language available that operates on the ASTs. In Spoofox, Stratego is used for transformations, where the terms and their constructors are part of the data language. Having concisely represented terms therefore also helps keeping transformations concise. In Python, it is more common to reason in terms of classes with attributes. Since all expression types have similar attributes, such as a list of subexpressions, it makes more sense to define one generic constructor.

Though the Minidom AST representation in the Python implementation is automatically generated, thanks to the Minidom library, this library can only be used for XML-based languages. It can be expected that for non-XML languages much more effort is needed to define an AST representation, which is also the reason why the implementation does not define IRs. In Spoofox, AST schemas can be defined for any textual language in an equally productive way.

Desugaring transformations. **Question.** Does it cost less code volume to define the desugaring transformations for OIL in Stratego compared to Python?

Method. We collect all desugaring transformations that are implemented in both the Spoofox and Python implementation and measure the SLOC used to implement them. Any SLOC called by the desugaring transformations, such as helper functions, are counted too.

Results. Table 4.2 shows the SLOC used to implement the desugaring transformations in both implementations. Any SLOC that are used for more purposes than one desugaring transformation are captured in the “Reused” row. The body of reused code in the Python implementation mainly consists of helper functions for traversal through the Minidom AST representation or for general simple transformations. The Stratego implementation has less reused code, because most of such traversals or simple transformations can be compactly expressed with the Stratego language and its standard library.

Artifact	Stratego	Python
<i>distribute-groups</i>	41	41
<i>auto-region</i>	16	31
<i>auto-super</i>	31	37
<i>auto-state</i>	127	94
<i>auto-value</i>	5	34
<i>auto-variable</i>	39	52
<i>auto-type</i>	26	31
<i>auto-init</i>	15	38
<i>auto-silent</i>	9	18
Reused	42	251
Total (Without reused)	309	376
Total (All)	351	627

Table 4.2: SLOC for the implementation of desugaring transformations that are defined in both the Spoofox and Python implementation of OIL.

Not every desugaring transformation in this table corresponds to one transformation in Stratego and one function in Python. Desugaring transformation *distribute groups* corresponds to two Stratego transformations and one Python function, *auto-region*, *auto-variable*, *auto-type* and *auto-init* correspond to one Stratego transformation and two Python functions, and *auto-state* corresponds to two Stratego transformations and three Python functions.

Analysis. In total, the desugaring transformations are implemented with Stratego with a factor of 0.56 SLOC compared to Python. On average, not counting reused SLOC, a desugaring transformation is implemented with Stratego with a factor of 0.82 SLOC compared to Python. This difference is mainly due to the fact that Stratego is specifically tailored for transformations, it allows one to define transformations in a more compact way. Stratego does this with ATerm as the main data format and with the core support for pattern matching.

We take the implementation of *auto-value* as an example. Due to ATerm as data format, one can create the resulting state node by writing the resulting term as the right-hand side of a transformation rule (Fig. 4.22 line 5), instead of creating the new node and text fields step by step (Fig. 4.26 lines 6-9). Due to the core support for pattern matching, checking whether the state has no value and assigning the name of the state to a variable can be done by just writing the `State` term with required subterms (such as `None()`) and variables (such as `name`) as the left-hand side of a transformation (Fig. 4.22 line 4), whereas in the Python implementation this is done in separate steps (Fig. 4.26 lines 4-5). For this example, the default support in Stratego for creating a fresh name (`newname` in Fig. 4.22 line 7) also helps significantly, as this is implemented explicitly in the Python implementation with a (not reused) function of 23 SLOC (called in Fig. 4.26 line 7).

Deviations from the average SLOC ratio that are more in favor of the Python

implementation are typically in case of transformations that require less local changes, such as *auto-region*, *auto-super*, *auto-state* and *auto-variable*, which introduce regions, zones, states and variables respectively, based on multiple sources of information in the AST. This is for a few reasons. First of all, it is not possible in Stratego to access the parent of a term directly. To access such information, a top-down traversal is needed that keeps track of previously encountered terms. Secondly, it is not straightforward in Stratego to store information globally to reuse later, which is for instance necessary to make sure that all names created are distinct. It is possible to use dynamic rules [100] for this, but the dynamic transformation behavior that these rules add makes it more difficult to understand how Stratego code executes when reading it. Lastly, the immutability of ATerm makes it impossible to store references to parts of the AST. First collecting information and then transforming this information does not have an effect on the resulting AST; instead, the information needs to be mapped back to the AST to then transform it, or information collection and transformation should be intertwined. In Python, these three restrictions of Stratego are less of a concern due to the freedom one has in a general-purpose programming language.

An example type of operation used in desugaring that is affected by the restrictions of Stratego above is finding the least common ancestor (LCA) of two areas. In the Python implementation, this is done by walking up the AST from the two areas using parent pointers until both paths cross. In Stratego, such a walk along parent pointers is not possible. Instead, the complete AST is traversed bottom-up in a recursive fashion, where for each area in the tree all descendant areas are collected. If the two areas for which the LCA needs to be found are in this collection for the first time, the LCA is found. An implementation for finding the LCA has similar SLOC between Stratego and Python, but the implementation in Python is reused between multiple transformations, while the implementation in Stratego is not as it is part of the basis of the desugaring transformation.

The only difference in functionality between the two implementations are some naming conventions for newly introduced elements. For instance, in *auto-value*, the Stratego implementation creates a name that is different from any name in the OIL specification, whereas the Python implementation creates a name that is different only from all literals in the OIL specification. This choice is sufficient, but requires one to specifically collect all literals, which contributes to 21 SLOC of *auto-value* in the Python implementation.

Conclusion. Although there are some restrictions to Stratego that hinder the conciseness of transformations defined in it compared to Python, on average Stratego requires less code volume compared to Python to define the desugaring transformations.

Discussion. As mentioned in the analysis, there are some restrictions to Stratego due to limitations on what it supports when compared to Python, such as not being able to directly access the parent of a term. We do not necessarily see these restrictions as points of improvement however. For instance, while the immutability of the AST may make it impossible to store references to

terms in the AST, which restricts the design of transformations, this does make sure that an AST cannot be transformed indirectly, which is beneficial for the understanding of Stratego code.

The measurements only consider the SLOC used to implement the functionality of the desugaring transformations, not the composition of them. In the Spoofox implementation, they are composed using the resilient staging framework. This framework is defined with 63 SLOC, which can be reused for any language and any transformation architecture. Creating a stage from a desugaring transformation then costs one stage transformation rule call, as shown in Fig. 4.23. The stages are then sequentially composed. Since the Python implementation does not implement resilient staging explicitly, it does not have this overhead.

4.7 STATIC SEMANTICS

In this section, we discuss the implementation of static semantics of OIL in Spoofox, which we consider to consist of name binding, typing and other well-formedness aspects, which together realize OF₃ (Input Correctness). Name binding and typing are implemented using NaBL₂. Well-formedness is mostly implemented with a collection of Stratego transformations that produce errors if the constraints are violated. In the transformation architecture of OIL, well-formedness checking occurs on the normalized IR, while name binding and typing occur on the desugared IR (see Fig. 4.20).

We describe how transformations and origin tracking are used to realize well-formedness checking and how cross-file and cross-language analysis over a collection of IDL and OIL files is realized, which relates to OF₄ (Language Interaction). Afterwards, we discuss how static semantics is realized in the Python implementation. We then evaluate Spoofox on productivity in the context of static semantics.

4.7.1 Well-formedness Checking

OIL specifications need to conform to well-formedness constraints. For example, each region should have at least one state. Although this particular example could have been enforced in the grammar, not all well-formedness constraints can be enforced in a grammar, or they lead to messy grammars. Also, by implementing these constraints manually, it is possible to generate better error messages than generic errors generated by the parser.

For some well-formedness constraints, such as illegal variable names and name distinctness, there is core support in SDF₃ and NaBL₂ respectively. Other constraints are checked by means of Stratego transformations, which transform an AST to a list of errors. Fig. 4.27 depicts a Stratego rule that implements a well-formedness constraint that says that every state must have a value. This check is used as the postcondition of the stage of desugaring transformation *auto-value*, see Fig. 4.24. If the check fails, the rule returns a list of errors, one for each state for which the check fails. Each error is a tuple that contains both the state term as well as the error message.

Although the constraint is defined on an IR — and thus not on the original

```

1 all-states-value =
2   collect-all(\
3     s @ State(_, None(), _, _) ->
4     (s, "State does not have a value")
5   \)

```

Figure 4.27: A Stratego code snippet that checks whether all states have a value.

```

1 init ^ (s) :=
2   ...
3   new s,
4   distinct/name D(s)/Module | error "Duplicate module" @NAMES.
5
6 [[ IDLModule(m,imps,defs) ^ (s) ]] :=
7   Module{m} <- s,
8   ...

```

Figure 4.28: A NaBL2 code snippet that checks duplication of module names.

parsed AST — Spoofox can associate the error with the original input syntax. This is thanks to origin tracking. The origin information of a term, that is created when a specification is parsed, is passed on when this term is transformed into another term. This makes the origin information directly accessible from the term within an error tuple. Spoofox can then use these error tuples to show the error to the user on the correct syntactical element in the editor.

4.7.2 Cross-file and Cross-language Analysis

The implementation of OIL and IDL in Spoofox involves static analysis that spans both multiple files and multiple languages. For instance, for a collection of multiple IDL files it should be checked whether there are no modules defined with the same name (cross-file analysis, within the same language). As mentioned in Section 4.3.2, transitions in OIL refer to operations defined in IDL files (cross-language analysis). We discuss how both forms of analysis are implemented using NaBL2.

Fig. 4.28 depicts an NaBL2 code snippet that enforces the cross-file constraint that no modules with duplicate names may exist across IDL files. Line 1 defines the `init` rule, which is where the analysis starts. NaBL2 is configured such that the scope `s` that is created in the `init` rule (line 3) is used as the initial scope for each IDL file. For IDL files, this scope is supplied to the rule for `IDLModule` (line 6), which adds a declaration of the module to this scope. By attaching the scopes of all IDL files to the single root node, all IDL modules are part of a single scope graph. The restriction that all module names are distinct is defined on line 4, where `D(s)/Module` defines the collection of all

```

1  [[ DESImportModule(m) ^ (s) ]] :=
2  Module{m} -> s,
3  Module{m} |-> decl | error "Module not found" @ m,
4  Module{m} <=== s.

```

Figure 4.29: A NaBL2 code snippet that imports IDL modules into OIL.

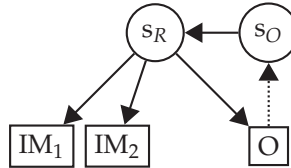


Figure 4.30: An abstract representation of a merged scope graph of an OIL specification (O) and two IDL modules (IM_i). Rectangles indicate declarations, circles indicate scopes. S_R is the root scope. The scope associated with the OIL specification (S_O), indicated with the dotted arrow, has the root scope as its parent scope, thereby making the IDL modules visible from the OIL specification.

Module elements reachable from scope s and `distinct/name` defines that no two elements in this set may have the same name.

Fig. 4.29 depicts an NaBL2 code snippet that specifies the cross-language importing of IDL modules into OIL specifications. First a reference to the module is added to the scope (line 2), after which it is checked whether the referenced module can be found (line 3), that is, whether a path from the reference to the declaration exists in the scope graph. Then in line 4 all declarations in the module are imported. More precisely, `Module{m} <=== s` makes all declarations that are visible in the scope on which `Module{m}` was declared visible in s .

Similar to how multiple IDL files share a single scope graph, the scope graphs of OIL files could conceptually be connected with those of IDL files to import the analysis for importing of Fig. 4.29. Fig. 4.30 depicts this. However, in Spoofox it is not possible to implement this directly. Spoofox only supports configuring NaBL2 to have analysis span multiple files of a single language, but not the files of multiple languages. This has been worked around by instantiating a single language artifact that accepts both IDL and OIL files. Although the implementation sources of IDL and OIL are organized in separate projects, there is no distinction anymore between an IDL and OIL language artifact; for IDL modules to be usable in OIL specifications, they have to be in files with the same `.oil` extension.

4.7.3 The Python Implementation

In the Python implementation, typing of expressions is implemented with a bottom-up recursive algorithm with a case distinction on the type of operator.

```

1 def extractModules(spec):
2     idlInfo = spec.generator.getIDLInfo()
3
4     rootModules = OrderedSet()
5     for _import in spec.getElements('import'):
6         moduleName = _import.getAttribute('module')
7         if moduleName in idlInfo:
8             rootModules.add(moduleName)
9         else:
10            spec.addAttributeValueError(_import, 'module', None, 'Module not
11            found in "idl_specs.txt" nor in the additional IDL include directory.')
    spec.rootModules = rootModules

```

Figure 4.31: A Python code snippet that imports IDL modules into OIL.

For some operators, such as equality, the process is repeated but with an expected type. Name resolution is partly done by the type checker and partly by separate functions. Well-formedness constraints are defined with separate functions.

See Fig. 4.31 for how name resolution of import statements is done in the Python implementation. On line 2, information on the IDL files is retrieved. This loads the relevant IDL files and creates a dictionary representing them, if this was not done already. Then on lines 5-7 the function iterates over all module names that appear in import statements. It checks on line 7 if this module name exists and if not, it reports an error (line 10). On line 11, the list of imported modules is stored in the OIL specification object for easy access.

Whereas the Spoofox implementation aborts the pipeline for any stage pre- and postcondition that fails, the Python implementation can do multiple steps before aborting. When to abort in case of errors is decided manually, by means of conditional return statements throughout the sequence of desugaring transformations and well-formedness checks. Errors can be shown in the web interface of the Python implementation on a textual representation of the OIL specification thanks to origin tracking. Syntactical elements with errors are highlighted in red and hovering over them shows an error message.

4.7.4 Evaluation

To evaluate the productivity of implementing static semantics, we look at a single evaluation point: the implementation of static semantics in OIL's implementations in Spoofox and in Python. We consider five static semantics artifacts: name binding, typing, well-formedness, error handling and error reporting.

Question. Does it cost less code volume to define the static semantics artifacts for OIL in Spoofox compared to Python?

Method. For the name binding, typing and well-formedness artifacts that are implemented in both Spoofox and Python, we measure how many SLOC were used to implement them. We also measure the SLOC used to abort when errors are found (error handling) and the SLOC used to show the errors

Artifact	Spoofax	Python
Name binding	128	233
Typing	81	257
Well-formedness	24	57
Error handling	15	29
Error reporting	19	37
Reused	208	298
Total (Without reused)	267	613
Total (All)	475	911

Table 4.3: SLOC for the implementation of name binding, typing and other well-formedness over the syntactic constructs that are defined in both the OIL and Python implementation.

to the user (error reporting). Any code called by name binding, typing and well-formedness definitions, such as helper functions, are counted too.

Results. Table 4.3 shows the SLOC used to implement name binding, typing and other well-formedness, as well as error handling and error reporting in both Spoofax and Python. Only name binding and typing over syntactic elements that are defined in both the Spoofax and the Python implementation are considered. Any SLOC that are relevant for more artifacts than just one out of name binding, typing and well-formedness are captured in the “Reused” row. All SLOC under Spoofax name binding, typing and reused are written in NaBL2. More specifically, NaBL2 code that only relates to creating and querying the scope graph corresponds to name binding and NaBL2 code that only relates to type definitions and type checking corresponds to typing; the rest corresponds to “Reused”. Well-formedness in Spoofax is a combination of SDF3, NaBL2 and Stratego. Error handling and error reporting are defined in Stratego.

To create a fair comparison, we have not counted any SLOC that produces functionality that is not in the other implementation. For the name binding and typing in Spoofax written in NaBL2 this meant that 88 SLOC was not counted. This 88 SLOC includes analysis of syntactic elements not implemented in Python and typing of elements that is not done in Python, such as areas and operations. For the name binding and typing by the type checker of the Python implementation 101 SLOC was not counted, which consists of analysis of syntactic elements and other checks not done in the Spoofax implementation. Since the well-formedness constraints are implemented as separate rules in Spoofax or functions in Python, we can measure them separately. Only a few well-formedness constraints have been measured, since many do not correspond well to any constraint in the other implementation.

Analysis. As Table 4.3 shows, name binding, typing and other well-formedness are implemented in Python in about double the SLOC compared to Spoofax. In general, the lower SLOC for Spoofax can be explained by the fact that the meta-DSLs that are used, especially NaBL2, are specifically made

for the implementation of these aspects.

Only looking at SLOC specific to name binding, the Spoofox implementation uses a factor of 0.55 SLOC compared to the Python implementation. In NaBL2, SLOC specific to name binding consist of creating and querying the scope graph. In Python, this involves reading in IDL files, creating classes for easy access to information in IDL files, checking name binding by querying this information and that of the OIL specification, and adding name binding information to the AST. The last two cause the main difference in SLOC between NaBL2 and Python. Checking name binding in NaBL2 is done by adding the reference to the scope graph and then checking for a path to the declaration as shown in lines 2-3 in Fig. 4.29. How this declaration is found does not need to be implemented explicitly, while in Python all declarations are retrieved manually to explicitly check whether the relevant declaration exists. Adding name binding information to the AST also needs to be explicitly implemented in Python, while this implicitly happens in NaBL2 by having a constraint rule for every term.

Looking only at typing-specific SLOC, the Spoofox implementation uses a factor of 0.32 SLOC compared to the Python implementation. For both implementations, most of the SLOC are in the typing of expressions. One of the main reasons that the Python SLOC is higher than the NaBL2 SLOC is that in Python there are some binary operators, such as equality and assignment, for which many case distinctions are defined based on the types of operands they can have. In NaBL2 these case distinctions are not necessary as they happen implicitly.

Looking only at well-formedness-specific SLOC, the Spoofox implementation uses a factor of 0.42 SLOC compared to the Python implementation. This is partly due to core support for some specific forms of well-formedness in Spoofox, such as rejecting specific variable names and checking for distinctness of names within a scope. Such checks only take 1 SLOC in SDF₃ and NaBL2 respectively, see Fig. 4.28 for an example, whereas in Python these require explicit traversal of the AST. Other well-formedness constraints in Spoofox are implemented in Stratego, for which the same productivity conclusions hold as for desugaring transformations (Section 4.6.5).

For error reporting and error handling, the Spoofox implementation uses about half the SLOC compared to the Python implementation. The difference in error handling SLOC is because the Spoofox implementation has a generic way of aborting pipelines built into the resilient staging framework, while in the Python implementation abort points are placed manually, which produces code duplication.

Both implementations support two types of error reporting: by means of highlights on the original specification or by means of a list of errors. For the first, the Spoofox implementation only needs minor general configuration, while the Python implementation explicitly locates and colors the syntactical elements in an HTML generator specific for XML-based languages. For the second, both implementations support a generic way of displaying the list of errors.

For both implementations, a large portion of the SLOC are reused. The reused SLOC in Spoofox is only relevant for static analysis: 174 SLOC consists of constraint rule declarations that define how name binding and typing information is related between a term and its subterms, and 34 SLOC is related to naming and importing modules. The reused SLOC in Python consists mainly of parts of the type checker that also involve variable reference resolution, as well as helper functions that retrieve information from the OIL and IDL specifications, which are also used for more purposes than only static semantics.

Conclusion. For static semantics, the Python implementation uses about twice the amount of SLOC compared to the Spoofox implementation, for the same functionality, with or without considering reused SLOC. This is mostly due to the fact that NaBL2 is specialized for name binding and typing and because of core support for specific types of well-formedness. Error handling and reporting can also be defined in a more concise and generic way. This shows that it costs less code volume to implement the static semantics artifacts for OIL in Spoofox compared to Python.

Discussion. Like with the concrete syntax definition, the name binding and type checking of IDL and OIL in the Spoofox implementation is split up into multiple NaBL2 modules, 21 in total. Unlike with the concrete syntax definition, NaBL2 files do not need import statements to compose them. All NaBL2 files in a project are deemed relevant and are collected automatically, so no SLOC is needed to compose NaBL2 modules. An exception to this is the composition of OIL-specific analysis with IDL analysis, which costs about 12 SLOC. This includes project configurations to export the IDL NaBL2 definitions as well as imports of IDL type signatures and files generated from the IDL NaBL2 definitions.

The workaround to make it possible to do name resolution between IDL and OIL files did not cost any extra NaBL2 SLOC. It did however cost 26 extra Stratego SLOC (not in Table 4.3) to define transformation rules that check whether the given AST is an IDL or OIL specification, which are needed at the beginning of end-to-end transformations.

A big reason for the conciseness of NaBL2 is that it is a declarative language, which means that one does not implement how the program executes. This can however also make it unpredictable how the NaBL2 analysis is executed. For example, when a reference of an integer type is used at a location where a boolean is expected, the type error could be reported on the declaration of the integer variable, while the error is expected on the reference. Although NaBL2 specifications can be annotated to indicate a preference for reporting the error on the declaration, this does not cover all cases.

The implementation of name binding and typing in Spoofox also automatically provides some editor services. When hovering over a syntactical element in the editor, its type is shown in a small text box. Also, navigating through a reference moves the cursor to the corresponding declaration, even if both are in different files. The Python implementation does not provide these editor services.

4.8 DYNAMIC SEMANTICS

OIL is a language for defining the behavior of control software. What the actual behavior is of an OIL specification is described by its dynamic semantics, which is formally defined in [94]. This semantics is implemented in Spoofox using Stratego with two code generators: one for verification (mCRL2) and one for execution (C++). These together realize OF5 (Multiple Targets).

Like OIL, mCRL2 is a language for describing system behavior, except that it is based on process algebra [89]. It also comes with a toolset [101], containing all kinds of model checking functionalities, such as checking properties and checking behavioral equivalence, as well as tools to simulate and visualize the behavior of an mCRL2 specification. With this translation from OIL to mCRL2, the functionality of the mCRL2 toolset can be indirectly used for OIL specifications as well. Some early results of this were already presented in [94].

To actually use an OIL specification to implement a software system, executable code needs to be generated. For that reason, a translation from OIL to C++ was implemented. This translation is inspired by the C++ generator in the Python implementation, which was already used for some systems in development at Canon Production Printing.

We highlight three parts of the implementation of these two translations in Spoofox: how the implementation of dynamic semantics is split into many projects, how static analysis results are queried for use in transformations and how configurability of a translation is handled. Afterwards, we discuss the implementation of these translations in the Python implementation. We then evaluate Spoofox on productivity in the context of dynamic semantics.

4.8.1 Division into Projects

As was already discussed in Section 4.6 and shown in Fig. 4.20, an OIL specification is first transformed to the semantic IR in the Spoofox implementation before it is transformed into mCRL2 or GPL code. For the sake of extensibility, the semantic IR, mCRL2 and GPL are all defined in separate projects, as well as the transformations between them. See Fig. 4.32 for the hierarchy of these projects, where the SEM project defines the semantic IR. All projects in this hierarchy are part of the Spoofox implementation except for the mCRL2 project, which already existed for other purposes.

The translation to GPL defined in OIL2GPL does not translate directly to C++, but to an intermediate representation called the GPL IR first instead. The GPL IR is a pseudo code representation defined in the GPL project with basic object-oriented imperative programming language constructs such as classes, methods, basic statements and expressions. The GPL project then defines a translation from the GPL IR to C++ files. The reasoning behind the creation of the GPL IR is to make it relatively simple to add translations to other general-purpose programming languages: only a transformation from the GPL IR to that programming language needs to be implemented.

See Fig. 4.33 and 4.34 for how the GPL IR splits up the transformation of an enum declaration to C++. In Fig. 4.33, an enum type declaration of

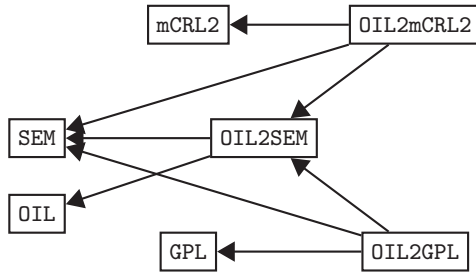


Figure 4.32: A project extension graph of projects used for code generation in the Spoofax implementation. Boxes correspond to projects. Arrows mean “extends”.

```

oil-sem2gpl-spec-enumdef(|specName):
  SEMEnumDef(type, items) ->
  GPLEnumDef(<SCOPEdtype(|specName, type)>, items)
  
```

Figure 4.33: The Stratego transformation of enum declarations from the semantic IR to the GPL IR.

```

1  gp12h: GPLProgram(_, ..., enumDefs, ...) ->
2  $[...
3  [<join-strings(|"\n\n")> <map(gp12h-enumDef)> enumDefs]
4  ...
5  ]
6
7  gp12h-enumDef: GPLEnumDef(name, items) ->
8  $[enum class [name] {
9  [<indent-text(|2)> <join-strings(|",\n")> items]
10 }];]
  
```

Figure 4.34: The Stratego transformation of enum declarations from the GPL IR to C++.

```

oil2mcrl2-type =
  nabl2-get-ast-type ; oil2mcrl2-map-type
oil2mcrl2-map-type : TInteger -> Int()
oil2mcrl2-map-type : TBool    -> Bool()

```

Figure 4.35: A Stratego transformation rule that transforms a term to one that represents its type in mCRL2.

the semantic IR is transformed to an enum type declaration of the GPL IR, which changes the type name using the name of the OIL specification and the original type name. In Fig. 4.34, for every enum type declaration (line 3), a C++ enum class is created (line 8-10). While most transformations in the Spoofox implementation of OIL are model-to-model transformations, the transformation from the GPL IR to C++ is a model-to-text transformation, as can be seen by the use of templates. This way, it is not necessary to define the C++ syntax in Spoofox.

4.8.2 Using Static Analysis Results

When NaBL2 name binding and typing have been applied on an AST, all terms in the AST are annotated with information that stores the results of the analysis. This information can then be used in Stratego transformations by means of specific transformation rules. For instance, the type of a term can be extracted with the rule `nabl2-get-ast-type`. See Fig. 4.35 for a (partial) definition of a transformation rule that uses this rule. Given a term with type information, such as a variable reference term, it returns a term that represents its type in the mCRL2 AST schema. The terms starting with T correspond to the type as annotated by NaBL2.

It is possible in NaBL2 to annotate the AST with more information than the default name binding and typing results. Scope graph nodes can be given *properties*, which can store any term. We use this for instance for retrieving the declaration of an enum type, defined in an IDL specification, when we generate code for an OIL specification that has a reference to this enum type. See Fig. 4.36 for an example, where the declaration of an enum type in an IDL specification is stored in its scope graph declaration node `Type{name}` with a property `decl` (line 3). This declaration node is then stored within the `TEnum` type `enum_ty` of the enum (line 4). Whenever a reference to this enum type in the OIL specification is encountered (line 8), we add a reference node to the scope graph (line 9) and try to resolve it (line 10) like discussed in Section 4.7.2. If successful, the `TEnum` type `ty` of the enum type reference (line 8) is inferred from the enum type declaration `d` by requiring that `d` also has type `ty` (line 11). Since property `decl` is stored within this `TEnum` type, the property becomes available in the context of the enum type reference in the OIL specification.

See Fig. 4.37 for a Stratego transformation in which the property is retrieved from an enum type reference, which is part of the translation from an OIL specification to mCRL2. First the type of the type reference is extracted (line 5),

```

1  [[ e@EnumDef(name, literals, ...) ^ (s) ]] :=
2      Type{name} <- s,
3      Type{name}.decl := e,
4      enum_ty == TEnum(Type{name}),
5      Type{name} : enum_ty !,
6      ... ]]
7
8  TypeRef [[ TypeReference(x) ^ (s) : ty ]] :=
9      Type{x} -> s,
10     Type{x} |-> d | error $[Type [x] not found] @ x,
11     d : ty.

```

Figure 4.36: Two NaBL2 code snippets: one that stores the declaration of an enum type inside its type (simplified) and one that defines type inference for type references.

```

1  sem2mcrl2-imported-enum-typedef:
2      ref -> ...
3  where
4      a := <nabl2-get-ast-analysis> ref;
5      TEnum(occ) := <nabl2-get-ast-type> ref;
6      EnumDef(type, items, _) := <nabl2-get-property(|a, "decl")> occ;

```

Figure 4.37: A Stratego transformation rule that uses the scope graph node stored in the type of an enum variable reference to obtain the declaration of the corresponding enum type.

```
oil2mcrl2-dyn-options(aux-vars) =
  (aux-vars < rules(mcrl2-aux-vars : t -> t) + rules(mcrl2-aux-vars : _
    -> <false>))
```

Figure 4.38: The creation of dynamic rules in Stratego for the auxiliary variables configuration.

```
sem2mcrl2-process-trans-pre =
  if mcrl2-aux-vars then sem2mcrl2-firedvar else sem2mcrl2-trans-pre end
```

Figure 4.39: An example Stratego transformation rule where the dynamic rule `mcrl2-aux-vars` is used.

after which the `decl` property is queried on the node within the type, which provides the enum type declaration (line 6). This enum type declaration can then be translated to one in mCRL2 (line 2, details not shown).

4.8.3 Configurability of the mCRL2 Generator

The translation to mCRL2 has a number of configuration options. Some of these options are mainly useful for debugging the translation during development, but other options result in a significantly different output. For instance, one option mainly used for debugging is whether to use auxiliary variables in the generated mCRL2 specification that help enhance its readability. Since these options change how the mCRL2 specification should look like, they configure the transformation that generates mCRL2. One way to implement this in Stratego is by passing the configuration information on as parameters of transformation rules. However, the more complex a transformation becomes and the more deeply nested this information is used, the more cluttered with such configuration parameters the transformation becomes. A solution in many languages would be to define global variables that hold this information, but Stratego does not support global variables.

Instead, *dynamic rules* [100] are used. A dynamic rule is a transformation rule that is created during transformation time, whose behavior can depend on the status of the transformation at that time. Such rules are used for configuration by creating a dynamic rule for each value of a configuration option, which always succeeds if the value was chosen, otherwise it always fails. For boolean configuration options one dynamic rule suffices. These rules can then be used wherever the differences between configurations have effect on the transformation, without having to pass anything on explicitly.

See Fig. 4.38 for a Stratego rule that creates the dynamic rule `mcrl2-aux-vars` for the auxiliary variables configuration, which is done just before the transformation from the semantic IR to mCRL2 is applied. The parameter `aux-vars` stores whether the user has chosen to introduce auxiliary variables. This parameter is then used in a ternary operator of the shape `s1 < s2 +`

```

localEnumTypes = spec.getLocalEnumTypes()
for tName in localEnumTypes:
    tgt.nl()
    tgt <<= f'enum class {Camel(tName)}'
    tgt <<= '{'
    tgt.indent()
    tgt <<= (f',{tgt.eol}{tgt.ind}') .join(localEnumTypes[tName])
    tgt.dedent()
    tgt <<= '};'

```

Figure 4.40: The Python transformation for enum declarations.

s3, which acts similarly to an if-then-else. If `aux-vars` is true, the dynamic rule `mcr12-aux-vars` is defined as a rule that always succeeds (`t -> t`), else as one that always fails (`_ -> <false>`). See Fig. 4.39 for an example where this dynamic rule is used during the transformation to mCRL2. If the user chose for the introduction of auxiliary variables, the precondition of a transition in the mCRL2 process should be represented with an auxiliary variable (`sem2mcr12-firedvar`), otherwise the full transition precondition is used (`sem2mcr12-trans-pre`).

4.8.4 The Python Implementation

The Python implementation also defines a translation to mCRL2 and a translation to C++. The translation to mCRL2 in the Python implementation was created during an exploratory study on the semantics of OIL and is therefore only a prototype. Compared to the Spoofox mCRL2 generator, the Python mCRL2 generator supports slightly fewer OIL language constructs and it can only generate mCRL2 code for single components, whereas the Spoofox mCRL2 generator can also generate mCRL2 for systems of components. On the other hand, the Python C++ generator has been maintained and refined for years and has been used to generate C++ for systems used in production. Compared to the Spoofox C++ generator, the Python C++ generator supports slightly more OIL language constructs and it is built to fit into Canon Production Printing's software base. This includes adherence to coding standards and a higher level of configurability of the generated C++ code, such as allowing multiple types of schedulers to execute the specification, which is not supported in the Spoofox C++ generator.

Both the Python mCRL2 generator and the Python C++ generator are defined in their own files in the Python implementation. Since the Python implementation does not have any explicit IRs, both generators directly transform the (desugared) OIL specification to the desired target. See Fig. 4.40 for an excerpt of the Python C++ generator that transforms an enum declaration to C++. First all declared enum types are collected from the desugared OIL specification (line 1), after which a C++ enum class is printed line by line for each enum type (lines 2-9).

Artifact	Spoofax	Python
mCRL2 generator	689	531
C++ generator	705	1321
Reused	508	369
Total (Without reused)	1394	1852
Total (All)	1902	2221

Table 4.4: SLOC for the implementation of the mCRL2 and C++ generator in both the Spoofax and Python implementation of OIL.

The code generators in Python also support the use of static analysis results and configurability. Static analysis results and properties are stored by dynamically adding new fields to the classes that represent terms. This information can then be accessed directly when needed. Configuration options are stored globally, which can be directly accessed from anywhere in the translation.

4.8.5 Evaluation

To evaluate the productivity of implementing dynamic semantics, we look at a single evaluation point: the implementation of code generation. More specifically, we look at the mCRL2 and C++ generators that are available in the Spoofax and Python implementation.

Question. Does it cost less code volume to define code generation for OIL in Spoofax compared to in Python?

Method. We measure the SLOC of the code generators used to transform a desugared and analyzed OIL specification to mCRL2 and to C++. Any SLOC called by the code generators, such as helper functions, are counted too.

Results. Table 4.4 shows the SLOC used to implement the mCRL2 and the C++ generator in both Stratego and Python. Any SLOC that are used for more purposes than one code generator are captured in the “Reused” row.

Because the exact differences in functionality (of generated code) between the two implementations and what SLOC attributes to these differences is very complex to measure, we decided to measure the SLOC of the code generators in full. This complexity is due to multiple factors. One is that structure of the code generators is very different: the code generators in Spoofax are split up into multiple transformations between IRs, while the code generators in Python do a direct translation from a (desugared) OIL specification to the target. Another is that code generators do not almost only differ in syntactic OIL constructs they support, as is the case for concrete syntax and static semantics, but also what they support in the functionality of the generated code, which is much more difficult to compare accurately.

The Spoofax implementation of the mCRL2 generator consists mainly of Stratego code from the OIL2mCRL2 project (689 SLOC). This code generator also uses the mCRL2 project (373 SDF3 SLOC), but since this project already existed outside the scope of our project, we do not include the SLOC measurements of this project in our results. The Spoofax implementation of the

C++ generator consists mainly of Stratego code from the OIL2GPL and GPL projects (389+151 SLOC) and SDF₃ code for defining the grammar of the semantic IR from the GPL project (165 SLOC). The reused code in the Spoofox implementation consists mainly of SDF₃ code for defining the grammar of the semantic IR from the SEM project (164 SLOC) and Stratego code for various helper transformations used by more than one code generator. The mCRL₂ and C++ generators in the Python implementation are both implemented in separate files. The shared Python code consists of helper functions that are used for both code generators.

Analysis. The Spoofox mCRL₂ generator uses a factor of 1.3 SLOC compared to the Python mCRL₂ generator. The Spoofox C++ generator uses a factor of 0.39 SLOC compared to the Python C++ generator. A big reason for the difference in SLOC ratio of the two code generators is the difference in maturity. The Python mCRL₂ generator and the Stratego C++ generator are prototypes that only implement basic code generation. The Stratego mCRL₂ generator and the Python C++ generator have more functionality and have been maintained extensively compared to their prototype counterpart.

Diving deeper into the code generators shows that a big difference between the Stratego and the Python implementation is in the use of IRs. In the Spoofox implementation the code generator consists mainly of model-to-model transformations. The actual target syntax is created using the pretty printer that is automatically generated from the syntax in case of the mCRL₂ generator, and using a model-to-text transformation from GPL in case of the C++ generator (see Fig. 4.34). In Python, no IRs are used: the target code is printed line by line while using the (desugared) Minidom AST of the OIL specification to collect information (see Fig. 4.40).

The use of IRs does come with the overhead of defining the IRs. Both IRs have been defined by means of SDF₃ grammars; the semantic IR uses 164 SLOC and the GPL IR uses 165 SLOC. These could have been implemented with fewer SLOC if implemented with signatures in Stratego like with the normalized and desugared IR, as only these signatures are necessary for the transformation. With 52 constructors for the semantic IR and 44 constructors for the GPL IR, the signatures could be implemented with 1 SLOC per constructor. The difference in SLOC compared to SDF₃ is mainly due to the syntax needing lexical elements, needing priority definitions and the definition of some constructors in SDF₃ being spread over multiple lines for better pretty printing. However, using SDF₃ for this does give the benefit of having a readable syntax and the automatic generation of a pretty printer, which has been proven useful when debugging transformations.

Concerning the comparison of using Stratego over Python for the actual transformation, the same benefits and downsides as for desugaring transformations hold here. Stratego's use of ATerm as data format and its core support for pattern matching helps writing transformations in a concise way, but its not possible to directly access the parent of a term and the immutability of ASTs makes it impossible to create (global) references to parts of an AST. Given the structure of the transformations for the code generators, where the input AST

is only used to collect information and the target AST is built up from scratch, these downsides have less of an effect here compared to desugaring, where information collection and transformation is done in the same AST. Specifically for code generators, there is another small downside of using Stratego over Python in the form of the use of properties. In the Python implementation, due to the mutability of the AST, properties can be added and extracted with a single operation. In the Spoofox implementation, as shown in Section 4.8.2, Fig. 4.37, multiple operations are necessary to retrieve the declaration of an enum type.

The reused Python code mainly consists of general helpers for information collection, AST traversal and code generation. The reused Stratego code mainly consists of transformations to and on the semantic IR, which include calculation steps necessary for the semantics of OIL, such as computing transition pre/postconditions. In the Python implementation, these calculation steps are done separately in both code generators, resulting in some code duplication. This code duplication could have been avoided by creating more shared helper functions, reducing the total amount of SLOC.

The modularity that comes with splitting up the code generators in the Spoofox implementation into multiple projects, as shown in Fig. 4.32, also comes with a cost in SLOC. To configure the projects such that they extend each other, 31 SLOC is used. Since the code generators in the Python implementation are defined in a single file, no SLOC is needed for anything similar.

Conclusion. To implement an mCRL2 and a C++ generator, the Spoofox implementation uses a factor of 0.86 SLOC compared to the Python implementation (a factor of 0.75 when not counting reused SLOC). With the differences in functionality between the Spoofox and Python code generators in mind, we cannot not draw a conclusion on the definition of code generators.

Discussion. The use of annotated information was one of the things that was most difficult to get working correctly. Because the information is stored on scope graph nodes instead of the terms itself, the information is not easily retrievable. With the NaBL2 interface for Stratego, it was not possible to extract the scope graph nodes that belong to a term directly from this term. Some ideas for NaBL2 properties, such as whether a variable reference refers to one declared in an OIL specification or an operation parameter, were never implemented due to this. The idea to put the scope graph node inside a type as described in Section 4.8.2 is actually more of a workaround, as the type of a term is easily retrievable with the NaBL2 interface. We are not sure whether this is an issue of the NaBL2 interface or of the lack of documentation on it. In Statix, the successor of NaBL2, properties are directly associated with terms instead of scope graph nodes, which alleviates this issue.

A benefit of the model-to-model approach with IRs is reusability of transformations. A good example of this is the definition of C++ methods. In the Stratego implementation, C++ methods are created by defining GPL methods first. Only a single transformation rule needs to be defined to translate a GPL method to a C++ method. In the Python implementation, the syntactical details of each method are repeated every time a new method is defined.

Another benefit of using IRs is that it is good for the extensibility of the implementation. Adding a Java generator to the Stratego implementation only requires a translation from GPL to Java in the GPL IR project, which reuses the transformation to the semantic IR and the transformation from the semantic IR to the GPL IR (389 out of 540 SLOC of the C++ generator in Table 4.4). In the Python implementation, a new transformation from a desugared OIL specification would need to be defined. This is assuming that the GPL IR is capable of representing all Java constructs that are necessary for the resulting output. If that is not the case, adjustments need to be made to the GPL IR and any transformation to and on it.

4.9 EVALUATION

In this section we summarize the main findings for our research question and we discuss their threats to validity.

4.9.1 Summary

RQ: *How does the productivity of implementing an industrial language in Spoofox compare to the productivity when using a GPL and available libraries?*

To answer this, we have measured and compared the code volume used to implement language engineering artifacts in the Spoofox and Python implementations of OIL. Both evaluated implementations are complete, in the sense that all five desired OIL features as described in Section 4.3.3 are realized, except for OF₁ (Multiple Syntaxes). OF₁ is not implemented in the Python implementation, which is why we only compared SLOC for OILXML. For concrete syntax, abstract syntax, and static semantics we compared artifacts produced by both implementations with similar functionality. For dynamic semantics we could not make a clear comparison between the Spoofox and the Python implementation due to the large difference of maturity of the mCRL2 and C++ generators of the two implementations.

For concrete syntax the Spoofox implementation uses a factor of 0.29 SLOC compared to the Python implementation. This difference is mainly caused by the fact that most concrete syntax artifacts are automatically generated from the SDF₃ grammar definition in Spoofox, while in Python they are manually implemented, though reusable for other XML-based languages. When not counting these reusable parts, the Spoofox implementation uses a factor of 1.81 SLOC instead compared to the Python implementation.

For abstract syntax we considered AST representations and desugaring transformations. Since the ASTs are represented very differently in both implementations, we could not derive an insight. Comparing the code volume for desugaring transformations, we found that the Spoofox implementation using Stratego uses a factor of 0.56 SLOC compared to the Python implementation (a factor of 0.82 SLOC when not counting reused SLOC). The difference is mainly due to Stratego's core support for pattern matching on ASTs, although

the immutability of ASTs in Spoofox can be inconvenient when specifying transformations.

For static semantics the Spoofox implementation uses a factor of 0.49 SLOC compared to the Python implementation. Especially NaBL2's declarative nature, where name binding and typing are defined by means of scope graph and constraint generation rules, helps with keeping the implementation concise.

In summary, for concrete syntax, desugaring transformations and static semantics, the code volume used in Spoofox was about a factor 0.5 or less compared to Python. This is mainly due to the availability of meta-DSLs that are tailored to implementing language development aspects and to generating editor services. When not counting reused SLOC, the results are somewhat more favorable for the Python implementation. Since the comparison is on two implementations covering similar functionality, the results are an indication that it is more productive to implement OIL in Spoofox than in Python.

4.9.2 Threats to Validity

We discuss threats to our study's construct, internal, and external validity. We discuss using code volume as proxy for productivity both as construct- and internal validity.

Construct Validity

Construct validity concerns to which extent our code volume measurements actually assess productivity. As threats to construct validity, we discuss using code volume per artifact as proxy for productivity and bias in artifact selection.

Code volume per artifact as proxy for productivity. Using code volume per artifact as a proxy for measuring productivity is a controversial measure [96, 97, 95, 98] and a threat to construct validity. Especially for measuring absolute productivity the measure is controversial, as many other factors could have influenced the effort it took to create an implementation. For example, developers can spend the majority of their time on program comprehension and only a small portion on writing code [102]. In general, to mitigate the threat of using code volume per artifact as a proxy, we use the code volume measurements to compare two implementations, not to derive absolute productivity numbers. Second, both implementations already existed before the evaluation, which counters the threat that one implementation could have been optimized in terms of code volume to get better evaluation results. Third, in each evaluation we aim to compare parts of both implementations that cover the same functionality.

A threat that remains is that implementing a DSL is not just about writing lines of code, but also about the time needed to understand how to do so with the implementation language(s) available. The average time per SLOC is influenced by the experience of the developer and the language that is used, e.g., Python is more commonly known than Spoofox and its meta-DSLs. Also, earlier experience with language engineering or compiler construction is beneficial. From our experience, especially NaBL2 requires considerable time to learn. The Master students that contributed to the project seemed to pick

up Stratego rather quickly.

We will now discuss using code volume as proxy for productivity in more detail for specific language aspect evaluations. In the concrete syntax evaluation, the original Spoofox and Python implementations did not cover the exact same syntactic languages. For example, this is due to the Spoofox implementation still containing some language constructs that have been removed from the Python implementation. To increase the fairness of our comparison, we have subtracted the lines of code for syntactical elements that are not present in the other implementation. In the Spoofox implementation, this was 31 out of 396 SLOC (7.8%). In the Python implementation, this was 46 out of 1306 SLOC (3.5%). Compared to the productivity comparison outcomes, these differences are inconsequential.

From the abstract syntax evaluation we take Fig. 4.26 as an example. The desugaring rule *auto-value* in the Python implementation could have been implemented using list comprehension to do multiple steps on the same line of code. This would reduce the lines of code but would also make the code more complex to understand. These threats are mitigated by the fact that both implementations have been created without the goal of evaluating them, let alone optimizing the lines of code, rather than with the goal of being correct and well-maintainable.

Bias in artifact selection. In our evaluations, we measure code volume for a selection of artifacts; our selection of artifacts could be biased. This raises the question how representative the selected artifacts are for the whole implementations and thereby is a threat to construct validity. Since for every language aspect the selected artifacts cover almost the whole implementation, we think this threat is negligible.

Next to the implementation of OIL in Spoofox's meta-DSLs and in Python, an implementation of a DSL also contains other code for, e.g., configuration and the build system. We have not included these in the measurements, which could make our measurements less representative for the whole implementations. From our experience, the code spent on configuration and build specification is so little that we do not expect the outcomes of our study to be different if they were included.

Both implementations contain code that is specific to some artifact and code that is reused for multiple artifacts. Some reusable code can even be used beyond OIL, which is especially the case for the implementation of concrete syntax in Python. Since reusability of code impacts productivity, we measure reused code separately and discuss how reusable the code is. When comparing both implementations, we compare both with and without reusable code.

Internal validity

Internal validity concerns to which extent our measurements actually represent the effect on productivity, and cannot be caused by other factors. For internal validity, we discuss using code volume per artifact as proxy for productivity, design decisions, confirmation bias, and experience of developers.

Code volume per artifact as proxy for productivity. In the static semantics evaluation, not much SLOC has been measured for well-formedness compared to name binding and typing, because not many well-formedness constraints were implemented in Spoofox and most of those that are, do not correspond well with constraints in the Python implementation. One of the main reasons for this is that well-formedness was not a high priority during the development of OIL in Spoofox. Therefore, we cannot give a strong indication regarding the productivity of implementing well-formedness.

Interdependence of implementations. Both implementations were not created entirely independently from each other. The Python implementation was already well maintained when we started with the Spoofox implementation. When developing the Spoofox implementation, the Python implementation was used to determine what should be implemented, for instance, which desugaring transformations are necessary and what the code resulting from the C++ code generator should look like. However, the Python implementation was not used to determine *how* things should be implemented in the Spoofox implementation. The meta-DSLs of Spoofox differ from Python so much that there is no clear translation from Python to a meta-DSL, or vice versa. Therefore, we believe that the SLOC measured in one implementation are independent of the SLOC measured in the other implementation.

Design decisions. During the implementation of a DSL, several design decisions are made that influence the implementation. Therefore, particular design decisions can have influenced the outcomes of our study. We have countered this threat by taking two implementations of the same language that are realized independent of our evaluation, i.e., the implementations already existed before this evaluation was started.

The question remains whether the conclusions could have been different given totally different design decisions. For the Spoofox implementation, we think different design decisions would not lead to very different conclusions, as Spoofox and its meta-DSLs steer design decisions, leaving little design decisions to the language engineer. Also, several design decisions that were made in the Spoofox implementation, such as using language composition and many modules for code organization, came with overhead increasing the counted SLOC. For the Python implementation, we think many design decisions could have been made very different, which can steer the implementation to use more or less lines of code, which is a threat to internal validity. Given the nature of our study, where we focus on a complex industrial case, we think this threat is justified. Although all the services of Spoofox could be re-implemented using Python and offered as reusable code, that is not what typically happens as it would be over-design from the perspective of developing a single language.

Confirmation bias. Some of the authors have contributed to the Spoofox and Python implementations of OIL, which raises a concern regarding confirmation bias. We have mitigated the risk of confirmation bias in the following ways, which prevents the possibility for authors to, during the study, change the implementations or steer evidence in a way that supports prior beliefs. First,

we have chosen a fixed version of the Spoofox and Python implementations of OIL from a moment before the SLOC measurements started. Second, while code measurements are conducted by a single author, at least one other author has checked these measurements. Third, the authors involved in the implementations of OIL had many discussions to ensure that code measurements cover those parts of the implementation to make comparisons as fair as possible.

Experience. Not all developers who worked on the Spoofox implementation were familiar with Spoofox and its meta-DSLs. Therefore, it could be that the meta-DSLs were not used optimally, and code is unnecessarily large at some points in the implementation. We do not expect this to have large impact on the outcomes of our study. For the Python implementation this is not much of an issue as it is a language (paradigm) that the developers were more experienced with.

External Validity

External validity concerns to which extent our findings are generalizable to other language workbenches, comparison to other GPLs, other DSLs, and other contexts. Our study focuses on a particular language workbench (Spoofox), a comparison with a particular GPL (Python), a particular DSL (OIL), and a particular context (the industrial context of Canon Production Printing). Therefore, it is unclear to what extent our findings also hold for other language workbenches, comparison to other GPLs, other DSL cases, and other contexts, as a specific case study is not easy to generalize. OIL's implementations in Spoofox and Python cannot be published due to confidentiality reasons, which hinders the reproducibility of our study.

Generalizability of Python. The Python implementation heavily relies on object-oriented programming and the availability of, e.g., parsing libraries. These aspects are not uncommon for other GPLs and therefore we expect that our findings can be similar for comparisons to other GPLs. Features that are more specific to Python, such as list comprehension, are rarely used in the Python implementation.

Generalizability of OIL. We do think our case is representative of industrial DSL development because OIL is a complex DSL with requirements specific to the industrial context. Still, OIL has specific characteristics that could be very different from other DSLs. Many DSLs only have one syntax, while OF1 required support for multiple syntaxes. Also, OIL is dependent on another language, IDL, following OF4, while DSLs are often self-contained. On the other hand, relating to OF3, OIL has rather simple typing and name-binding rules. We think that desugaring transformations (OF2) and code generation (OF5) are rather common for DSLs, though the structure of the transformations and generators may differ, and some DSLs may be interpreted instead.

4.10 DISCUSSION

While our evaluation in the previous section is based on conclusions drawn from our quantitative analyses, in this section we discuss aspects of our case study that are of a qualitative nature. First, we discuss the strengths and weaknesses of Spoofox that we have experienced. Second, we list the lessons learned from our study. Finally, we suggest an engineering agenda for Spoofox. In the engineering agenda for Spoofox, we also discuss if and how the weaknesses of Spoofox we have encountered are improved upon in the next version of Spoofox.

4.10.1 *Spoofox's Strengths*

We list several aspects that worked out well in using Spoofox.

Meta-languages suitable for OIL. The meta-languages that are used (SDF₃, NaBL₂ and Stratego) all offered sufficient support for implementing OIL's concepts. SDF₃ was sufficient for the implementation of OILXML's grammar and enabled rapid prototyping of OILDSL. The name binding and typing features of OIL and IDL could be specified in NaBL₂ using the scope graph model. OIL's transformations and code generators could be implemented using Stratego.

Modular language implementation. All meta-languages supported modular language implementation in the sense that implementations could be split up in modules that could be composed or reused. This was beneficial to the Spoofox implementation in many ways. For example, reusing SDF₃ modules for shared expression grammar prevented the need to define duplicate grammar rules for the four input languages (see Section 4.5.2). Stratego allows modular and composable definitions of transformations. In particular, Stratego enabled us to implement additional AST schemas and the resilient staging framework, which helped in creating a modular transformation architecture. Lastly, there is little overhead in creating new (composed) languages, which enabled us to easily add an extra language (IDL-OIL-TEST-DSL) specific for testing scenarios of multiple IDL and OIL specifications in isolation.

IDE support. Spoofox derives several editor services automatically for language implementations: parsing, AST inspection, syntax highlighting, syntax error recovery, showing type information, reference resolution, execution of analysis and transformations on file changes, execution of transformations on user request, and marking errors on the specifications. This made it feasible for us to realize an IDE for OIL. The ability to offer a DSL with a user-friendly IDE is beneficial for the adoption of DSLs in an industrial environment such as Canon Production Printing.

Language testing. SPT was useful for testing the implementations of IDL and OIL and to maintain implementation correctness while evolving the languages. SPT supports testing of several (isolated) aspects of the languages such as parsing, name resolution, and typing, as well as end-to-end tests for testing the translations to C++ and mCRL₂. Testing helps in obtaining a reliable language

implementation and validation during language evolution. For example, when adding or changing functionality to OIL, tests help to ensure other functionality is maintained.

Integration support. Spoofox contains three features for integrating a language implementation within a software ecosystem. First, Stratego offers a Java API which makes it possible to manually implement a transformation rule in the general-purpose language Java, which also enables integration of external tools. This Java API has been used to integrate the SAT solver Z₃ for static analysis to optimize generated C++ code [91]. This could also enable automated integration with the mCRL₂ toolset, i.e., by automatically calling mCRL₂ in a transformation. Second, Spoofox languages can be built outside Eclipse using the Maven or Gradle build systems. This should make it possible to integrate OIL in larger software builds such as continuous integration (CI) or production builds, which is relevant for software development at, e.g., Canon Production Printing. Third, Spoofox offers a Java API (named Spoofox Core) which enables to integrate parts of a Spoofox language implementation such as the parser or transformations within the Java ecosystem. Potentially, these features in combination can enable the integration into an existing industrial software ecosystem.

4.10.2 Spoofox's Weaknesses

We list several aspects that did not work out well in using Spoofox.

Limited portability. Portability concerns to what extent software can be used in different environments. Spoofox currently only provides full support in Eclipse as the IDE for language development and limited support for IntelliJ IDEA. This lack of portability limits the practical applicability opportunities of the language workbench. For example, at Canon Production Printing, software engineers mainly use the Visual Studio IDE, which is currently not supported by Spoofox. Although Spoofox does support integrating parts of a language implementation outside Eclipse using the Java API, the meta-DSLs are not available as independent libraries, hindering integration with other tooling.

Building and runtime performance. The language development experience in Spoofox is hindered by long build times and long response times after changes, sometimes blocking you for minutes. Although it is workable, it does not conform to the response times expected from interactive systems. The editing experience is non-concurrent, e.g., while a build is busy and one changes a file, the build first has to finish before the changed file gets reanalyzed. If a project consists of multiple subprojects, all subprojects have to be built manually one by one in the correct order, because automatic derivation of the correct build order is lacking. This especially becomes cumbersome in a project such as OIL that consists of 14 subprojects, which together take about 16 minutes to build on a company-provided laptop. When changes are made, the projects that are affected by the changes need to be rebuilt. Especially in an industrial context this is a problem, as costly time of engineers is spent on building projects rather than actual development.

Cross-language static analysis. Spoofox with NaBL2 does not offer native support for merging scope graphs of languages to realize language composition on the static semantics level. Conceptually, language composition on the static semantics level using scope graphs boils down to merging the root node of two languages' scope graphs. In practice, this required a workaround by merging the language definitions of IDL and OIL in one language project which accepts both IDL and OIL specifications (see Section 4.7.2). This is a workaround that could be resolved if Spoofox would offer coupling separately-defined languages by merging their scope graphs. The languages could then live next to each other, with their own file extensions, and only interact on scope graphs during static analysis.

Lack of static checking and debugging in NaBL2 and Stratego. The language development experience in NaBL2 and Stratego sometimes was hindered by the limited static checking of specifications in the meta-DSLs. As a result, it often occurs that errors made in a specification are only encountered during execution. For example, a Stratego strategy can fail on getting an incompatible type of term as input which could have been statically checked if strategies were typed. Also, no interactive debugging support for transformations is available. When transformations fail, stack traces are reported without references to the source code with line numbers. This is problematic in an industrial context as it makes engineers spend more time on debugging.

Using static analysis in transformations. Using the NaBL2 analysis results in transformations is cumbersome because low-level querying of the scope graph is required for general operations such as finding a declaration given a reference (see Section 4.8.2). The API is also sparsely documented, which makes it unclear how the API should be used. Spoofox could improve here by offering abstractions for common static analysis querying patterns.

Language evolution and refactoring. Evolving a language implementation in Spoofox can lead to cumbersome situations. For example, when IDL and OIL change, all specifications written in IDL and OIL have to be migrated manually. If the signature of a term changes, many Stratego transformations may need to be migrated as well. This has occurred in practice, for instance when area type "scope" was renamed to "zone". Applying the change of a name throughout the implementation involves intensive searching and replacing. Spoofox could be improved by adding support for cross-project and cross-meta-DSL refactorings in language definitions, similar to how modern IDEs support this.

Fine-grained testing. SPT mostly supports end-to-end testing of language implementations, whereas it was often desired to test individual parts of the implementation in a more fine-grained manner. For example, it was only possible to test desugaring transformations with SPT by defining tests that, given an OIL specification, generate the normalized IR, apply the desugaring transformation on it, and then transform it back to the original syntax. The success of this test does not only depend on the desugaring transformation, but also on the transformations between the textual OIL specification and the normalized IR. It would be useful if, in SPT, one could write a test for

a particular transformation rule or strategy for an input directly written as `ATerm`, essentially unit testing a small part of a transformation.

Editor actions for configurable code generators. As discussed in Section 4.8.3, some code generators have a number of configuration options. A user can pick values for these configuration options when selecting a code generator in editor action menus, which are defined using ESV (Spoofox's meta-DSL for defining editor services, see Section 4.2.6). However, it is not possible in ESV to reuse (sub)menus; every (sub)menu and menu item must be defined explicitly. When adding a new configuration option with n possible values, n times more editor actions need to be defined, which makes the size of the ESV file exponential in the number of configuration options.

4.10.3 Lessons Learned

We list our most important lessons learned from implementing OIL in the industrial context of Canon Production Printing both using Python and using Spoofox 2:

1. The meta-DSLs in Spoofox are just like DSLs limited to a certain domain, and it is not unheard of that we end up at the edges of this domain. For us, the meta-DSLs in Spoofox have been sufficient in the industrial context. Except for a few practical workarounds, we have experienced no limitations in implementing concrete syntax (with `SDF3`), abstract syntax (with `Stratego`), static semantics (with `NaBL2` for typing and name binding and with `Stratego` for well-formedness checking), and dynamic semantics (with `Stratego`).
2. The biggest limitations of Spoofox 2 are not in the functional aspects of meta-DSLs, but in their non-functional characteristics, e.g., slow build and response times, limited documentation, limited portability, and limited static checking of meta-DSL specifications.
3. Choosing XML and Python is a viable engineering choice in the absence of a language workbench. XML is a good choice for an effective implementation of concrete syntax for a DSL if dependence on external tools is undesired. Therefore, this is a simple alternative to using a language workbench with a penalty of roughly twice the code size and half of the editor features, as well as a penalty in the user-friendliness of the language.
4. A main benefit of DSLs is the multiplicative factor: from a single specification in a DSL, multiple backends can be targeted or multiple artifacts can be generated. This multiplicative factor is essential for the effectiveness of meta-DSLs used to implement DSLs: a single specification in a meta-DSL can generate multiple language processing artifacts and editor services. For instance, from an `SDF3` grammar, not only a parser is generated, but also an AST schema, a pretty printer, origin tracking and editor services.
5. Separate meta-DSLs for separate language implementation aspects lead to a clear separation of concerns, making it effective to define and maintain language aspects within those concerns. From our experience, the fundamental design decision of Spoofox to have clearly separated meta-DSLs seems to be

working well.

6. Specifications written in Spoofox's meta-DSLs can have high reusability and extensibility, by decomposition into modules, but this can come with a considerable cost in terms of code to compose the modules. However, since this code almost only consists of declaring and importing modules, we recommend to use Spoofox's meta-DSLs in a modular way.

4.10.4 Spoofox Engineering Agenda

Based on our experiences from implementing OIL with Spoofox 2, we suggest the following improvements to make on the language workbench to increase its adoptability in industry. We have presented these items to the Spoofox development team and incorporated their responses with respect to if and how these items have improved upon in Spoofox 3.

Portability. By making Spoofox available to more IDEs, more developers could make use of it in their IDE of choice. When companies have a policy on which IDEs engineers should use, not supporting such IDEs can block Spoofox from being adopted. Potentially, adding Language Server Protocol (LSP) support can help in improving Spoofox's portability; in principle the support needs to be implemented once but will make Spoofox available to all IDEs that support LSP. Although Spoofox 3 is not more portable out of the box (it supports Eclipse, Gradle, and a command line interface), it features a fundamentally different architecture than Spoofox 2. By supporting static rather than dynamic loading of languages, it is easier to extend Spoofox 3 with support for other IDEs or LSP. Custom language integrations are also easier to make, as languages can be packaged as Java libraries. In addition to languages developed with Spoofox, it would also be useful to offer the meta-DSLs as libraries, as that would ease integration with other tooling and allows the meta-DSLs to reach wider audiences.

Language build system. Several improvements can be made to the language build system provided by Spoofox to improve the development experience: improving build times (e.g., by further incrementalizing builds), automatically building a project that consists of multiple subprojects in the right order based on dependencies, and automatically checking whether exports and imports of files between projects are valid such that errors are detected early and do not require trial and error to debug. Spoofox 3 improves on all these aspects with the introduction of the PIE (Pipelines for Interactive Environments) [75] build system which features fully incremental builds for implementing both Spoofox 3 itself as well as languages developed with Spoofox 3. This is also relevant to debugging Stratego 2 code: with quick enough compilation round-trip, print debugging becomes much more viable.

Runtime performance. Improving the response times after changes in Spoofox would improve the development experience, such that less time during development is spent on waiting. In Spoofox in Eclipse, concurrent editor actions would improve the development experience; currently, e.g., when a build is busy, changes to other files are only picked up after the build finishes. Spoofox

3 with incrementalization using PIE improves runtime performance of both language builds as well as responsiveness of interactions in the IDE. With the introduction of PIE, runtime performance is not better for all implementation aspects as, e.g., the same SDF₃ parser generation is used which itself is not incrementalized.

Cross-Language Static Analysis. Spoofox could be improved by supporting cross-language static analysis by making it possible to merge the scope graphs of two separately defined languages, as also described by Zwaan [103]. In Spoofox 2 this was deemed virtually impossible to implement. Thanks to the new architecture of Spoofox 3, it supports the implementation of cross-language static analysis, which should support the OIL and IDL case. Cross-meta-language static analysis was one of the goals of Zwaan [103], but have not yet been materialized.

Static Checking in meta-DSLs. Improved static checking in Spoofox's meta-DSLs would enhance the language development experience by reducing the need for trial and error. The next version of Spoofox partly improves on these aspects, in the meta-DSLs Stratego 2 and Statix (successor to NaBL2). Stratego 2 introduces gradual typing [70] and Statix comes with static checks on its specifications.

Stratego Debugging. Spoofox only supports debugging of Stratego transformations by adding debug transformation rules that print information to the console. It would be beneficial for development with Stratego to be able to step through a transformation interactively, while showing the values of local variables and the term that the transformation is applied on. Spoofox 3 and Stratego 2 do not yet support debugging of transformations.

Integrating Static Analysis with Transformations. An improved API for using static analysis results in transformations can make transformation definitions more simple. In NaBL2's successor, Statix [72], some issues are alleviated already. For instance, in Statix properties are defined on terms directly instead of on scope graph nodes, which makes querying them straightforward.

Documentation. Improved documentation will help engineers new to Spoofox to learn and adopt the tool, without having to learn from experiences from others or by experimentation.

Unit Testing Stratego. It would be desired to have core support for unit-testing Stratego transformation rules in SPT, by supplying an input term, a transformation, and an expected result term. Especially for large and modular transformation architectures, the restriction of only testing end-to-end transformations makes it difficult and cumbersome to test individual parts of the transformation architecture.

ESV-Stratego integration. A better integration between ESV and Stratego could make the definition of editors services more simple. For example, supporting parameterized Stratego transformations in ESV would avoid redundant definitions. In the current state of ESV, the size of an ESV specification grows

exponentially in the number of configuration options available for an end-to-end transformation.

4.11 RELATED WORK

We discuss related work on evaluating language workbenches (and tools to develop DSLs in general) and other process languages such as OIL.

Most research on Spoofox focuses on the language workbench's fundamentals, with artificial languages as examples. An exception of this is Visser's case study on the development of WebDSL [33], which discusses language design and implementation for a DSL in the domain of web programming. The paper highlights the DSL development process and how the different aspects of this process can be covered by the meta-DSLs of Spoofox. This study uses Spoofox version 1, the predecessor of the Spoofox version used in our study. Several discussion sections cover DSL engineering evaluation criteria focusing on the process and the language that is produced, not on the tools for developing the language (SDF + Stratego), language engineering paradigms, or language engineering challenges. Therefore, this work does not evaluate Spoofox itself or how it compares to not using a language workbench. Hamey and Goldrei [104] reported on their experiences of using SDF and Stratego compared to using traditional techniques. They found that the Stratego toolset enabled easy implementation with opportunity of enhancing the language and improving performance of generated code, compared to the implementation using traditional techniques.

Canon Production Printing uses modeling languages across various engineering disciplines [17]. Schindler et al. describe how the company envisions the use of models during the complete life cycle of printers to address the challenges of efficiently performing continuous innovation with sustainable quality. The MPS language workbench is selected as one of the core technologies to develop custom DSLs that can interconnect the models from different engineering disciplines and tools. The authors find that using modeling approaches has advantages: users only have to learn a single tool, multiple models can be generated from a single tool, and one point of maintenance is needed instead of multiple. They also encountered challenges in using MPS: steep learning curve, lack of full-fledged DSL models in MPS for commodity languages such as C++, existing parsers or grammars are not immediately reusable, and performance can be undesirably low.

Voelter et al. [6] report on their experiences on using MPS for the development of mbeddr, a large set of languages and extensions of the C language that targets embedded software development. This work is, to our knowledge, the largest evaluation of a language workbench, spanning a case that involved around 10 person years of development effort in an industrial setting. Whereas our work is centered around evaluating productivity, the paper by Voelter et al. is centered around five topics concerning the use of MPS: language modularity, notational freedom and projectional editing, mechanisms for managing complexity, performance and scalability issues, and consequences for the development process. The authors draw generally positive conclusions and

indicate various places for improvement as well.

Broccia et al. [105] state that although the quantitative aspects of language workbenches are often discussed in literature (e.g., the evaluations and comparisons by Erdweg et al. [8]), the evaluation of comprehensibility of the meta-languages used in language workbenches are typically neglected. The authors evaluate the Neverlang [26] language workbench on four aspects. First, the comprehensibility of programs in Neverlang in terms of users' effectiveness and efficiency in code comprehension tasks. Second, the relationship between comprehensibility and users' working memory capacity. Third, to which extent users consider the language workbench acceptable in terms of perceived ease of use, usefulness, and intention to use. Fourth, how comprehensibility relates to the degree of acceptance of the language. The study suggests that users' working memory capacity may be related to the ability to comprehend Neverlang programs. Effectiveness and efficiency do not appear to be related to an increase of users' acceptance variables. We believe more studies like these can be useful for getting a better understanding of how language workbenches are perceived and what influences their adoption.

Klint et al. [57] found that using DSL tools (ANTLR, OMeta, Microsoft "M") improve the maintainability of language implementations by comparing several implementations of the same DSL both with and without the use of DSL tools; the implementations without DSL tools use GPLs (Java, JavaScript, C#). The evaluation considers parsing, static analysis, and transformation. The results suggest that DSL tools increase maintainability of DSL implementation compared to using GPLs. The work is similar to our work by comparing implementations of a DSL using GPLs to implementations using tools specific for DSL development. The work by Klint et al. differs from our work in the sense that they compare six implementations instead of two, the DSL tools do not cover aspects of language engineering such as deriving IDEs, and they focus on maintainability instead of productivity.

Åkesson et al. [106] report on their experiences on the implementation of a Modelica compiler using JastAdd [107] compiler tool. In particular, an aim is to achieve extensibility of the compiler, which led to the choice of using the declarative attribute grammar approach provided by JastAdd. They illustrate how existing design strategies for a Java compiler implemented using JastAdd could be reused for advanced features of the Modelica compiler. The authors show complex semantic rules can be implemented in a compact and modular manner. Given the 9 man-months of development time that was spent on creating the implementation, they find that JastAdd is very well suited for rapid compiler development.

Basten et al. present a language engineering case study on a Rascal implementation of Oberon-o [108], focusing on how the language can be implemented in a modular way. Oberon-o consists of four language levels where each succeeding level is implemented as an extension of the preceding level, supported by Rascal's modularity features. The implementation used less than 1500 SLOC which includes the implementation of parsing, name and type analysis, desugaring, transformation, and compilation to C. Additionally, they

found directions for improvement for Rascal.

Zarrin et al. [109] report on their experiences of introducing a DSL for material flow analysis using Microsoft DSL tools. Their motivation for using the DSL is to enable domain experts to evolve existing software to fulfill new requirements. The authors report that the DSL tools were mature enough to develop a complete DSL. Drawbacks include redundantly having to define semantics for simulation and code generation, the visualization of the metamodel is difficult to understand, and being limited to graphical notation.

Other implementations exist of DSLs like OIL for the specification of behavior. For instance, the language Dezyne developed by the company Verum⁹ can be used to define system behavior and its implementation in Guile¹⁰ includes multiple code generators [110]. The Comma framework¹¹ contains a collection of languages and tools to define and analyze the signatures and behavior of interfaces and is implemented using Xtext, which also supports many code generators [111]. BPMN¹² is a UML-like graphical language for modeling business processes maintained by the Object Management Group, implemented using MOF (Meta Object Family), with XSD for static semantics and XSLT¹³ for dynamic semantics [112]. SystemC¹⁴ is a language for simulating event-driven concurrent processes, defined as a subset of C++ with predefined classes and functions, which makes it possible to reuse much of the already existing analysis and editor services for C++.

4.12 CONCLUSIONS

In this chapter, we have presented an industrial case study on language engineering with the Spoofox language workbench. In summary, the contributions of this chapter are:

- An evaluation of whether Spoofox's original claims — on making language development, compared to not using a language workbench, more productive — stand when realizing the implementation of a complex industrial language such as OIL.
- Lessons learned on implementing OIL using Spoofox in the industrial context of Canon Production Printing.
- Strengths, weaknesses, and an agenda for future engineering on Spoofox.

We found that Spoofox and its meta-DSLs SDF₃, NaBL₂ and Stratego were adequate for implementing OIL. It was possible to implement every OIL feature using the meta-DSLs. Several workarounds were needed, such as for the interaction between IDL and OIL when it comes to static analysis, but these could still be implemented within Spoofox.

⁹<https://www.verum.com/>

¹⁰<https://www.gnu.org/software/guile/>

¹¹<https://comma.esi.nl/>

¹²<https://www.bpmn.org/>

¹³<https://www.w3.org/TR/xslt-30/>

¹⁴<https://systemc.org/>

In our evaluation, we found indications that it is more productive to implement a complex DSL with a language workbench compared to not using a language workbench. We did this by comparing the code volume (in SLOC) of two implementations of OIL, one using Spoofox and one using Python, which both already existed before the evaluation. The evaluation shows that the Spoofox implementation used fewer SLOC compared to the Python implementation, while offering more editor features. This is relevant in an industrial setting because it enables to develop a full-featured IDE with less code.

Naturally, our evaluation is not without threats to its validity. The use of SLOC as metric for productivity is contested. For instance, there can be much variance in what a line of code defines. We do feel that the results on code volume per artifact are an indication for higher productivity with Spoofox compared to Python, as the analyses show considerable differences in SLOC for artifacts with the same functionality and because both implementations were created before we had the intent to evaluate them. Since our evaluation is done for a single use case, it is difficult to generalize our findings to other workbenches, languages and contexts. Therefore, we call for more studies on applications of language workbenches in practice. This is relevant because it will help industrial language engineers decide when and how to use language workbenches.

In our study, we have primarily focused on evaluating and comparing productivity. Still, we were able to make several observations for other concerns such as modularity and maintainability of language implementations, both positive and negative. For example, the ability to easily extend SDF3 definitions and Spoofox projects benefits modularity and the ability to generate multiple artifacts from a single source benefits maintainability. On the other hand, the inability to merge scope graphs of different languages hinders modularity and the steep learning curve of NaBL2 hinders maintainability. Since concerns such as the modularity and maintainability of language implementations are important for developing DSLs in industry, we encourage more studies that evaluate language workbenches in detail on dimensions other than productivity.

Although Spoofox was suitable for implementing OIL, we see several areas of improvements. These are mainly in the practical use of the language workbench, such as limited portability, slow build and response times, and limited documentation. For the meta-DSLs we see the following opportunities for improvement: supporting cross-language static analysis, improving the API for using static analysis results in transformations, supporting unit testing, and improving the integration of Stratego in the definition of editor services. Several of these improvements have been included in the next version of Spoofox.

Based on our study, we provide the following advise.

- *For industrial language engineers:* Use a language workbench for developing DSLs especially if a user-friendly editor for the languages is desired; not doing so leads to “reinventing the wheel”, which can cost considerable effort.
- *For industrial language engineers:* When IDE support is not required, using,

e.g., off-the-shelf parser generators and a GPL could be a valid engineering choice for implementing the concrete syntax of a DSL, as the drawbacks of a language workbench may outweigh the benefits.

- *For language workbench developers:* Focus on the practical aspects of language workbenches such as portability, usability, and documentation to improve adoptability.

Conclusion

The underlying goal of this dissertation is to improve our understanding of how DSLs developed with language workbenches can impact industrial software engineering. Rather than taking a broad perspective, we took a deep dive into two idiosyncratic cases of DSL development at Canon Production Printing using the Spoofox language workbench. On the one hand, it is inherent to this approach that threats to external validity remain for our findings. On the other hand, the approach allowed us to extensively account for internal threats to validity, enabling us to produce valuable contributions to the language engineering community in both academia as well as industry. We summarize these contributions as follows:

DSL Creation We have designed and implemented two external DSLs in the context of Canon Production Printing using Spoofox: CSX and OIL. CSX is a DSL for the domain of configuration space exploration of digital printing systems, for which the development of a DSL was motivated by tackling complexity in a way that was found to be infeasible using traditional techniques. OIL is a DSL for modeling and implementing control software behavior. Its development using Spoofox was motivated by an existing implementation of OIL in XML and Python being difficult to advance and involving “reinventing the wheel”.

Industrial Evaluation We have evaluated both DSLs in the industrial context of Canon Production Printing. We evaluated CSX in terms of domain coverage, accuracy, and performance. For OIL, we evaluated how the productivity of implementing a DSL in Spoofox compares to the productivity when using a GPL and available libraries.

Lessons Learned We have collected lessons learned on developing a DSL with a constraint-solving backend and on developing OIL using both Python and Spoofox.

In the rest of this final chapter, we discuss our answers to the research questions, reiterate our lessons learned, and provide directions for future work.

5.1 RESEARCH QUESTIONS

We answer the overarching research question by answering the sub-questions.

RQ-CSX: *How does a DSL with a constraint-solving backend impact the development of control software for digital printing systems with respect to domain coverage, accuracy, and performance?*

We have investigated this research question using the CSX case study (Chapter 2 and Chapter 3). In this case study, we have developed a DSL with a constraint-solving backend for the development of control software for digital printing systems.

We have evaluated the domain coverage of CSX using think-aloud co-design sessions, in which the developer of CSX and a domain expert from Canon Production Printing together modeled a realistic printing system. We found that CSX was suitable for covering such a realistic digital printing system, although coverage for aspects such as grouping and ordering of sheets can still be improved.

We have evaluated the accuracy of CSX by testing for correctness and completeness. For correctness, we test that the configurations that are found for a device correspond to the device's limitations. For completeness, we test that all configurations that are possible in a device are found. The tests build confidence in the accuracy of CSX by testing all features and a subset of feature interactions at least once. Still, there can be untested feature interactions that are not handled correctly.

We have evaluated the performance of CSX using benchmarking on realistic scenarios of configuration space exploration, measuring translation and solving times. We have observed that the performance is acceptable for interactive usage (within the order of seconds) in several scenarios. However, performance is unpredictable, because for seemingly similar scenarios the solving can also time out.

CSX impacts the development of control software fundamentally in the sense that a DSL is used instead of a GPL. This enables the use of constraint solving, which in turn enables the realization of environments for configuration space exploration that are automatic and accurate. Given our positive evaluation of CSX's domain coverage, accuracy, and performance, CSX can provide its benefits for realistic printing systems.

Achieving configuration space exploration that is automatic and complete is an improvement over the existing development approach at Canon Production Printing. This existing approach was based on heuristics for finding configurations. Due to the complexity of control software, the heuristics-based implementations were not always accurate and they were difficult to maintain for a large variety of printing systems. Therefore, the CSX approach improves the quality of the printing systems by tackling the complexity that hinders improving quality in the existing development approach. This is emphasized by the additional value that CSX brings with its support for finding optimal configurations. The industrial value of CSX has been confirmed by the patent that records the underlying invention.

Our evaluation built confidence that CSX is useful for application in practice. Evaluation on a wider range of printing systems and scenarios of configuration space exploration could reveal limitations in the domain coverage and performance of the current version of CSX. We provided several ideas for improving domain coverage and performance to counter such limitations.

RQ-OIL: *How does the productivity of implementing an industrial language in Spoofox compare to the productivity when using a GPL and available libraries?*

We have investigated this research question using the OIL case study (Chapter 4). In this case study, we compared an implementation of OIL in Spoofox with an implementation of OIL in Python with XML syntax. We compared the two implementations on four language implementation aspects: concrete syntax, abstract syntax, static semantics, and dynamic semantics. For the comparison, we used lines of code as a proxy for productivity. For concrete syntax, abstract syntax, and static semantics, we found that the Spoofox implementation used about half of the lines of code compared to the Python implementation. Since the comparison is on two implementations covering similar functionality, the results are an indication that it is more productive to implement OIL in Spoofox than in Python. This is mainly due to the availability of meta-DSLs in Spoofox that are tailored to implementing these aspects and to generating editor services. For dynamic semantics, the difference in maturity between the two implementations was too large to draw any conclusions.

Our findings are relevant for Canon and comparable high-tech industries because they provide insight into the productivity of implementing a DSL using language workbenches. We showed how Spoofox was suitable for implementing a complex industrial language, and how the Spoofox implementation realized more features with less code compared to the Python implementation. These findings are important for industrial language engineers in deciding when and how to use language workbenches. Based on our analysis we expect that DSLs developed with language workbenches can improve language engineering productivity, which can reduce the cost of developing a DSL and thereby also improve the opportunity of DSLs for a positive return on investment.

Our evaluation considered a single language workbench, a single DSL, and a single dimension of software engineering (productivity), within a particular industrial context. The threats to the validity of our findings are in particular related to the generalizability to other language workbenches, DSLs, dimensions of software engineering (e.g., maintainability), and industrial contexts. Therefore, we call for more studies on the application of language workbenches in practice. This is relevant because it will further help industrial language engineers decide when and how to use language workbenches.

5.2 LESSONS LEARNED

In the CSX case study, we collected the following lessons learned on developing and applying a constraint-based DSL in an industrial context:

1. The Spoofox language workbench and the MiniZinc constraint modeling language (and compatible solvers) took care of much of the “heavy lifting” in realizing CSX. This enabled us to tackle complexity and improve functionality in software engineering for a complex domain by allowing us to mostly focus on the domain and language design.

2. A systematic approach to DSL evaluation is useful for communicating about a DSL in an industrial context. Concrete evaluation criteria for the use of a DSL help in the discussion to explain to people who have no experience with DSLs to understand what is required for a DSL to be applied in practice. Finally, the evaluation criteria guide decision-making regarding the adoption of the technique.
3. Starting to use a DSL in practice has a big impact on the software engineering process with dependencies on external tooling and having language engineering resources available for both language development and language maintenance. Therefore, the benefits of adopting a DSL need to be large to outweigh the corresponding investment.
4. The conceptual power of CSX is amplified by its IDE. The CSX IDE gives helpful insight into the behavior of models by featuring interactive validation of tests and debugging through inspection of configurations. This helped us to try out alternative language designs, leading to an iterative language design process.
5. It is a crucial language design decision to have types being defined in a language itself — instead of embedding a fixed set of domain objects in the language — which enables flexibility in modeling by iteratively including more detail in models. In CSX, this enabled experimenting with different representations of objects from the printing domain without changing CSX itself.
6. A high level of abstraction and domain-specific constructs such as in CSX are necessary to make constraint-based modeling accessible. Still, switching to the constraint-based programming paradigm can be challenging for developers who have no experience with constraint programming or with declarative programming at all.

We collected the following lessons learned from developing OIL in the industrial context of Canon Production Printing using both Python and Spoofox:

1. The meta-DSLs in Spoofox are just like DSLs limited to a certain domain, and it is not unheard of that we end up at the edges of this domain. For us, the meta-DSLs in Spoofox have been sufficient in the industrial context. Except for a few practical workarounds, we have experienced no limitations in implementing concrete syntax (with SDF₃), abstract syntax (with Stratego), static semantics (with NaBL₂ for typing and name binding and with Stratego for well-formedness checking), and dynamic semantics (with Stratego).
2. The biggest limitations of Spoofox 2 are not in the functional aspects of meta-DSLs, but in their non-functional characteristics, e.g., slow build and response times, limited documentation, limited portability, and limited static checking of meta-DSL specifications.
3. Choosing XML and Python is a viable engineering choice in the absence of a language workbench. XML is a good choice for an effective implementation of concrete syntax for a DSL if dependence on external tools is undesired.

Therefore, this is a simple alternative to using a language workbench with a penalty of roughly twice the code size and half of the editor features, as well as a penalty in the user-friendliness of the language.

4. A main benefit of DSLs is the multiplicative factor: from a single specification in a DSL, multiple backends can be targeted or multiple artifacts can be generated. This multiplicative factor is essential for the effectiveness of meta-DSLs used to implement DSLs: a single specification in a meta-DSL can generate multiple language processing artifacts and editor services. For instance, from an SDF₃ grammar, not only a parser is generated, but also an AST schema, a pretty printer, origin tracking and editor services.
5. Separate meta-DSLs for separate language implementation aspects lead to a clear separation of concerns, making it effective to define and maintain language aspects within those concerns. From our experience, the fundamental design decision of Spoofox to have clearly separated meta-DSLs seems to be working well.
6. Specifications written in Spoofox's meta-DSLs can have high reusability and extensibility, by decomposition into modules, but this can come with a considerable cost in terms of code to compose the modules. However, since this code almost only consists of declaring and importing modules, we recommend to use Spoofox's meta-DSLs in a modular way.

5.3 IMPLICATIONS & FUTURE WORK

For software language engineering researchers. Together with the work by Voelter et al. on the evaluation of the MPS language workbench [6], our evaluation provides empirical evidence on the application of DSLs and language workbenches in practice. However, these evaluations are limited in scope. We discuss three implications of this limitation. First, our work on evaluating Spoofox has been limited to productivity. Other claims made on Spoofox such as supporting a high degree of modularity are relevant to evaluate as well within industrial contexts, as modularity can enable reuse and extensibility and thereby improve language engineering. Furthermore, it is relevant to assess the usability of the usage environments generated by Spoofox, e.g., in terms of performance (providing feedback quick enough for interactive usage) and whether the provided editor services are on par with state-of-the-art usage environments for GPLs. Second, our evaluation of Spoofox focused on two particular cases in a specific industrial context. Validation of our results on a larger corpus of DSLs and in more industrial contexts is needed. Third, all available language workbenches vary in characteristics, and therefore evaluation of a wider range of language workbenches is needed.

For industrial language engineers. Industrial language engineers can use our findings to decide when to apply DSLs developed with language workbenches. At Canon Production Printing, there are several possibilities for bringing the work on CSX and OIL further into practice. For CSX, evaluation of domain coverage on a wider range of printing systems is needed to get a better

understanding of the DSL's effectiveness and scalability. Performance could potentially be improved by, e.g., using domain-specific information from CSX to instruct solvers in their search for solutions. Ultimately, we envision CSX as a language that could also be used by domain experts such as mechanical engineers, in which, e.g., the usability of the language and maintainability of the models would be of vital importance. This requires evaluating and possibly improving the usability of the CSX usage environment for domain experts. For OIL, the Spoofox implementation demonstrates several improvements to the existing implementation using Python, but it remains future work how these improvements can find their way into practice. As adopting a new language workbench for production usage is non-trivial in an industrial setting, CSX and OIL might not find their way into practice through Spoofox. In that case, the underlying ideas and improvements of CSX and OIL could be ported to a different implementation.

For language workbench developers. For Spoofox we found that the theory and concepts underlying the language workbench are more than adequate for creating industrial DSLs such as CSX and OIL. We found that Spoofox should especially improve on non-functional characteristics to increase its usability in industry. Based on the OIL case study, we provided a detailed engineering agenda for Spoofox. Many of our suggestions involve engineering that does not necessarily contribute to research. As Spoofox is currently developed and maintained primarily by researchers, it would be useful to have dedicated engineers working on the non-functional aspects of Spoofox. Individual language engineering components such as parser generation with SDF₃ could be made available as, e.g., libraries to improve the practical applicability of these components. We believe that Spoofox and other language workbenches are valuable tools that can positively impact software engineering, and we encourage efforts that bring their value further into practice.

Bibliography

- [1] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-Specific Languages: An Annotated Bibliography”. In: *SIGPLAN Notices* 35.6 (2000), pp. 26–36. DOI: 10.1145/352029.352035.
- [2] Martin Fowler. *Domain-Specific Languages*. Addison Wesley, 2010.
- [3] Andrzej Wasowski and Thorsten Berger. *Domain-Specific Languages: Effective modeling, automation, and reuse*. Springer, 2023.
- [4] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. “Satisfiability Modulo Theories”. In: *Handbook of Satisfiability - Second Edition*. Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, 2021, pp. 1267–1329. DOI: 10.3233/FAIA201017.
- [5] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and how to develop domain-specific languages”. In: *ACM Computing Surveys* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892.
- [6] Markus Voelter, Bernd Kolb, Tamás Szabó, Daniel Ratiu, and Arie van Deursen. “Lessons learned from developing mbeddr: a case study in language engineering with MPS”. In: *Software and Systems Modeling* 18.1 (2019), pp. 585–630. DOI: 10.1007/s10270-016-0575-4.
- [7] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005.
- [8] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. “Evaluating and comparing language workbenches: Existing results and benchmarks for the future”. In: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47. DOI: 10.1016/j.cl.2015.08.007.
- [9] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.
- [10] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.
- [11] Arie van Deursen and Paul Klint. “Little languages: little maintenance?” In: *Journal of Software Maintenance* 10.2 (1998), pp. 75–92. DOI: 10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5.

- [12] Federico Tomassetti and Vadim Zaytsev. “Reflections on the Lack of Adoption of Domain Specific Languages”. In: *STAF 2020 Workshop Proceedings: 4th Workshop on Model-Driven Engineering for the Internet-of-Things, 1st International Workshop on Modeling Smart Cities, and 5th International Workshop on Open and Original Problems in Software Language Engineering co-located with Software Technologies: Applications and Foundations federation of conferences (STAF 2020), Bergen, Norway, June 22-26, 2020*. Vol. 2707. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 85–94.
- [13] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. “Assessing the State-of-Practice of Model-Based Engineering in the Embedded Systems Domain”. In: *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*. Vol. 8767. Lecture Notes in Computer Science. Springer, 2014, pp. 166–182. DOI: 10.1007/978-3-319-11653-2_11.
- [14] Markus Voelter. “Programming vs. That Thing Subject Matter Experts Do”. In: *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings*. Vol. 13036. Lecture Notes in Computer Science. Springer, 2021, pp. 414–425. DOI: 10.1007/978-3-030-89159-6_26.
- [15] T. Stahl and Markus Völter. *Model-Driven Software Development*. New York: Wiley, 2005.
- [16] Markus Voelter. “Fusing Modeling and Programming into Language-Oriented Programming - Our Experiences with MPS”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I*. Vol. 11244. Lecture Notes in Computer Science. Springer, 2018, pp. 309–339. DOI: 10.1007/978-3-030-03418-4_19.
- [17] Eugen Schindler, Hristina Moneva, Joost van Pinxten, Louis van Gool, Bart van der Meulen, Niko Stotz, and Bart Theelen. “JetBrains MPS as Core DSL Technology for Developing Professional Digital Printers”. In: *Domain-Specific Languages in Practice: with JetBrains MPS*. Springer, 2021, pp. 53–91. DOI: 10.1007/978-3-030-73758-0_3.
- [18] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. “Supporting the understanding and comparison of low-code development platforms”. In: *46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portoroz, Slovenia, August 26-28, 2020*. IEEE, 2020, pp. 171–178. DOI: 10.1109/SEAA51224.2020.00036.
- [19] *Is a DSL suitable for you?* URL: <https://voelter.de/doyouneedone.html> (visited on 2024-02-14).

- [20] Lennart C. L. Kats and Eelco Visser. “The Spoofox 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*. Reno/Tahoe, Nevada: ACM, 2010, pp. 444–463. DOI: 10.1145/1869459.1869497.
- [21] *Jetbrains Meta Programming System*. URL: <https://www.jetbrains.com/mps>.
- [22] *Xtext*. URL: <https://www.eclipse.org/Xtext>.
- [23] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. “EASY Meta-programming with Rascal”. In: *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*. Vol. 6491. Lecture Notes in Computer Science. Springer, 2009, pp. 222–289. DOI: 10.1007/978-3-642-18023-1_6.
- [24] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. “MontiCore: a framework for the development of textual domain specific languages”. In: *30th International Conference on Software Engineering (ICSE 2008), Companion Volume*. Leipzig, Germany: ACM, 2008, pp. 925–926. DOI: 10.1145/1370175.1370190.
- [25] Benoît Combemale, Olivier Barais, and Andreas Wortmann. “Language Engineering with the GEMOC Studio”. In: *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*. IEEE, 2017, pp. 189–191. DOI: 10.1109/ICSAW.2017.61.
- [26] Edoardo Vacchi and Walter Cazzola. “Neverlang: A framework for feature-oriented language development”. In: *Computer Languages, Systems & Structures* 43 (2015), pp. 1–40. DOI: 10.1016/j.cl.2015.02.001.
- [27] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. “The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge”. In: *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*. Vol. 8225. Lecture Notes in Computer Science. Springer, 2013, pp. 197–217. DOI: 10.1007/978-3-319-02654-1_11.
- [28] Mikhail Barash. “Vision: the next 700 language workbenches”. In: *SLE '21: 14th ACM SIGPLAN International Conference on Software Language Engineering, Chicago, IL, USA, October 17 - 18, 2021*. ACM, 2021, pp. 16–21. DOI: 10.1145/3486608.3486907.

- [29] Fred D. Davis. “Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology”. In: *MIS Quarterly* 13.3 (1989), pp. 319–340.
- [30] Mark van den Brand. “A personal retrospective on language workbenches”. In: *Software and Systems Modeling* 22.3 (2023), pp. 847–850. DOI: 10.1007/s10270-023-01101-9.
- [31] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Néron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. “A Language Designer’s Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs”. In: *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH ’14, Portland, OR, USA, October 20-24, 2014*. ACM, 2014, pp. 95–111. DOI: 10.1145/2661136.2661149.
- [32] Eelco Visser. *A Brief History of the Spoofox Language Workbench*. URL: <https://eelcovisser.org/blog/2021/02/08/spoofox-mip/> (visited on 2023-12-21).
- [33] 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*. Vol. 5235. Lecture Notes in Computer Science. Braga, Portugal: Springer, 2007, pp. 291–373. DOI: 10.1007/978-3-540-88643-3_7.
- [34] Danny M. Groenewegen. “WebDSL: Linguistic Abstractions for Web Programming”. base-search.net (fttudelft:oai:tudelft.nl:uuiid:fbocc4b7-a67b-474b-9570-96ebo54a39ec). PhD thesis. Delft University of Technology, Netherlands, 2023.
- [35] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. “Design Science in Information Systems Research”. In: *MIS Quarterly* 28.1 (2004), pp. 75–105.
- [36] Jasper Denkers, Louis van Gool, and Eelco Visser. “Migrating custom DSL implementations to a language workbench (tool demo)”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. ACM, 2018, pp. 205–209. DOI: 10.1145/3276604.3276608.
- [37] Jasper Denkers, Marvin Brunner, Louis van Gool, and Eelco Visser. “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*. Vol. 13085. Lecture Notes in Computer Science. Springer, 2021, pp. 423–442. DOI: 10.1007/978-3-030-92124-8_24.
- [38] Jasper Denkers, Marvin Brunner, Louis van Gool, Jurgen J. Vinju, Andy Zaidman, and Eelco Visser. “Taming complexity of industrial printing systems using a constraint-based DSL: An industrial experience report”. In: *Software: Practice and Experience* (2023). DOI: 10.1002/spe.3239.

- [39] Olav Bunte, Jasper Denkers, Louis van Gool, Jurgen J. Vinju, Eelco Visser, Tim Willemse, and Andy Zaidman. "OIL: an Industrial Case Study in Language Engineering with Spoofax". In: *Software and Systems Modeling* (2024). DOI: 10.1007/s10270-024-01185-x.
- [40] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. "MiniZinc: Towards a Standard CP Modelling Language". In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007, pp. 529-543. DOI: 10.1007/978-3-540-74970-7_38.
- [41] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. "The MiniZinc Challenge 2008-2013". In: *AI Magazine* 35.2 (2014), pp. 55-60.
- [42] 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*. Vol. 12310. Lecture Notes in Computer Science. Springer, 2020, pp. 1-23. DOI: 10.1007/978-3-030-58768-0_1.
- [43] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. "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.
- [44] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "A Theory of Name Resolution". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 205-231. DOI: 10.1007/978-3-662-46669-8_9.
- [45] Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. "A constraint language for static semantic analysis based on scope graphs". In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 2016, pp. 49-60. DOI: 10.1145/2847538.2847543.
- [46] Arie van Deursen, Paul Klint, and Frank Tip. "Origin Tracking". In: *Journal of Symbolic Computation* 15.5/6 (1993), pp. 523-545.
- [47] 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*. Vol. 8137. Lecture Notes in Computer Science. Springer, 2013, pp. 182-196. DOI: 10.1007/978-3-642-40561-7_13.

- [48] Markus Voelter. “The Design, Evolution, and Use of KernelF - An Extensible and Embeddable Functional Language”. In: *Theory and Practice of Model Transformation - 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-26, 2018, Proceedings*. Vol. 10888. Lecture Notes in Computer Science. Springer, 2018, pp. 3–55. doi: 10.1007/978-3-319-93317-7_1.
- [49] Markus Voelter, Sergej Koscejev, Marcel Riedel, Anna Deitsch, and Andreas Hinkelmann. “A Domain-Specific Language for Payroll Calculations: An Experience Report from DATEV”. In: *Domain-Specific Languages in Practice: with JetBrains MPS*. Springer, 2021, pp. 93–130. doi: 10.1007/978-3-030-73758-0_4.
- [50] Daniel Jackson. “Alloy: a lightweight object modelling notation”. In: *ACM Transactions on Software Engineering Methodology* 11.2 (2002), pp. 256–290. doi: 10.1145/505145.505149.
- [51] Emina Torlak and Daniel Jackson. “Kodkod: A Relational Model Finder”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedi*. Vol. 4424. Lecture Notes in Computer Science. Springer, 2007, pp. 632–647. doi: 10.1007/978-3-540-71209-1_49.
- [52] 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*. ACM, 2019, pp. 46–61. doi: 10.1145/3359591.3359726.
- [53] Emina Torlak and Rastislav Bodík. “Growing solver-aided languages with rosette”. In: *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*. ACM, 2013, pp. 135–152. doi: 10.1145/2509578.2509586.
- [54] Jan C. Dageförde and Herbert Kuchen. “A compiler and virtual machine for constraint-logic object-oriented programming with Muli”. In: *Journal of Computer Languages* 53 (2019), pp. 63–78. doi: 10.1016/j.col.2019.05.001.
- [55] Karl Anders Ericsson and Herbert Alexander Simon. “Protocol analysis: Verbal reports as data, Rev”. In: (1993).
- [56] Kevin Leo and Guido Tack. “Debugging Unsatisfiable Constraint Models”. In: *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*. Vol. 10335. Lecture Notes in Computer Science. Springer, 2017, pp. 77–93. doi: 10.1007/978-3-319-59776-8_7.

- [57] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. “On the impact of DSL tools on the maintainability of language implementations”. In: *Proceedings of the of the Tenth Workshop on Language Descriptions, Tools and Applications, LDTA 2010, Paphos, Cyprus, March 28-29, 2010 - satellite event of ETAPS*. ACM, 2010, p. 10. DOI: 10.1145/1868281.1868291.
- [58] Arjan de Roo, Hasan Sözer, Lodewijk Bergmans, and Mehmet Aksit. “MOO: An architectural framework for runtime optimization of multiple system objectives in embedded control software”. In: *Journal of Systems and Software* 86.10 (2013), pp. 2502–2519. DOI: 10.1016/j.jss.2013.04.002.
- [59] Meinte Boersma. *Business-Friendly DSLs*. Manning, 2024.
- [60] Arie van Deursen, Jan Heering, and Paul Klint. *Language Prototyping: An Algebraic Specification Approach*. Vol. 5. AMAST Series in Computing. World Scientific, 1996. DOI: 10.1142/3163.
- [61] Vaclav Pech. “JetBrains MPS: Why Modern Language Workbenches Matter”. In: *Domain-Specific Languages in Practice: with JetBrains MPS*. Springer, 2021, pp. 1–22. DOI: 10.1007/978-3-030-73758-0_1.
- [62] Moritz Eysholdt and Heiko Behrens. “Xtext: implement your language faster than the quick and dirty way”. In: *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. ACM, 2010, pp. 307–309. DOI: 10.1145/1869542.1869625.
- [63] Mark Van den Brand, Arie van Deursen, Paul Klint, Steven Klusener, and Emma van der Meulen. “Industrial applications of ASF+ SDF”. In: *International Conference on Algebraic Methodology and Software Technology*. Springer. 1996, pp. 9–18.
- [64] Danny M. Groenewegen, Zef Hemel, Lennart C. L. Kats, and Eelco Visser. “WebDSL: a domain-specific language for dynamic web applications”. In: *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*. ACM, 2008, pp. 779–780. DOI: 10.1145/1449814.1449858.
- [65] Danny M. Groenewegen, Elmer van Chastelet, and Eelco Visser. “Evolution of the WebDSL runtime: reliability engineering of the WebDSL web programming language”. In: *Programming’20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*. ACM, 2020, pp. 77–83. DOI: 10.1145/3397537.3397553.
- [66] Daco Harkes and Eelco Visser. “IceDust 2: Derived Bidirectional Relations and Calculation Strategy Composition”. In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. DOI: 10.4230/LIPIcs.ECOOP.2017.14.

- [67] Daco Harkes, Elmer van Chastelet, and Eelco Visser. “Migrating business logic to an incremental computing DSL: a case study”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. ACM, 2018, pp. 83–96. DOI: 10.1145/3276604.3276617.
- [68] Eelco Visser. “Syntax Definition for Language Prototyping”. PhD thesis. University of Amsterdam, 1997.
- [69] Gabriël Konat. “Language-Parametric Methods for Developing Interactive Programming Systems”. PhD thesis. Delft University of Technology, Netherlands, 2019. DOI: 10.4233/uuid:03d70c5d-596d-4c8c-92da-0398dd8221cb.
- [70] Jeff Smits and Eelco Visser. “Gradually typing strategies”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*. ACM, 2020, pp. 1–15. DOI: 10.1145/3426425.3426928.
- [71] Jeff Smits, Gabriël Konat, and Eelco Visser. “Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System”. In: *Programming Journal* 4.3 (2020), p. 16. DOI: 10.22152/programming-journal.org/2020/4/16.
- [72] Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. “Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020). DOI: 10.1145/3428248.
- [73] Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. “Incremental type-checking for free: using scope graphs to derive incremental type-checkers”. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (2022), pp. 424–448. DOI: 10.1145/3563303.
- [74] Jeff Smits, Guido Wachsmuth, and Eelco Visser. “FlowSpec: A Declarative Specification Language for Intra-Procedural Flow-Sensitive Data-Flow Analysis”. In: *Journal of Computer Languages* 57 (2020), p. 100924. DOI: 10.1016/j.co-la.2019.100924.
- [75] Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. “PIE: A Domain-Specific Language for Interactive Software Development Pipelines”. In: *Programming Journal* 2.3 (2018), p. 9. DOI: 10.22152/programming-journal.org/2018/2/9.
- [76] Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. “Language-parametric static semantic code completion”. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA (2022), pp. 1–30. DOI: 10.1145/3527329.
- [77] Mark G. J. van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. “Efficient annotated terms”. In: *Software: Practice and Experience* 30.3 (2000), pp. 259–291. DOI: 10.1002/(SICI)1097-024X(200003)30:3<3C259::AID-SPE298%3E3.0.CO;2-Y.

- [78] Jan Willem Klop. “Term Rewriting Systems: From Church-Rosser to Knuth-Bendix and Beyond”. In: *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, July 16-20, 1990, Proceedings*. Vol. 443. Lecture Notes in Computer Science. Springer, 1990, pp. 350–369.
- [79] Noam Chomsky. “Three models for the description of language”. In: *IRE Transactions on Information Theory* 2.3 (1956). DOI: 10.1109/TIT.1956.1056813.
- [80] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. “Declarative specification of template-based textual editors”. In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. ACM, 2012, pp. 1–7. DOI: 10.1145/2427048.2427056.
- [81] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. “Disambiguation Filters for Scannerless Generalized LR Parsers”. In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 143–158. DOI: 10.1007/3-540-45937-5_12.
- [82] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. “Scopes as types”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018). DOI: 10.1145/3276484.
- [83] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. “Building Program Optimizers with Rewriting Strategies”. In: *Proceedings of the third ACM SIGPLAN international conference on Functional programming*. Baltimore, Maryland, United States: ACM, 1998, pp. 13–26. DOI: 10.1145/289423.289425.
- [84] Krzysztof Czarnecki and Simon Helsen. “Feature-based survey of model transformation approaches”. In: *IBM systems journal* 45.3 (2006), pp. 621–645.
- [85] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. “Testing domain-specific languages”. In: *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*. ACM, 2011, pp. 25–26. DOI: 10.1145/2048147.2048160.
- [86] Olav Bunte, Louis C. M. van Gool, and Tim A. C. Willemse. “Formal verification of OIL component specifications using mCRL2”. In: *STTT* 24.3 (2022), pp. 441–472. DOI: 10.1007/s10009-022-00658-y.
- [87] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. “Language composition untangled”. In: *International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012*. ACM, 2012, p. 7. DOI: 10.1145/2427048.2427055.

- [88] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmänn, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.
- [89] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.
- [90] Mark Frenken, Tim AC Willemse, Louis van Gool Océ, Olav Bunte, and Jasper Denkers. “Code generation and model-based testing in context of OIL”. MA thesis. Master’s thesis, Eindhoven University of Technology, 2019.
- [91] Tom Buskens. “Optimizing the code generator for OIL”. MA thesis. Master’s thesis, Eindhoven University of Technology, 2021.
- [92] Samuël Noah Voogd, Kousar Aslam, Louis van Gool, Bart Theelen, and Ivano Malavolta. “Real-Time Collaborative Modeling across Language Workbenches - a Case on JetBrains MPS and Eclipse Spoofox”. In: *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10-15, 2021*. IEEE, 2021, pp. 16–26. DOI: 10.1109/MODELS-C53483.2021.00011.
- [93] Louis van Gool. “Formalising interface specifications”. PhD thesis. Eindhoven University of Technology, 2006.
- [94] Olav Bunte, Louis C. M. van Gool, and Tim A. C. Willemse. “Formal Verification of OIL Component Specifications using mCRL2”. In: *Formal Methods for Industrial Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings*. Vol. 12327. Lecture Notes in Computer Science. Springer, 2020, pp. 231–251. DOI: 10.1007/978-3-030-58298-2_10.
- [95] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. “A SLOC counting standard”. In: *Cocomo ii forum*. Vol. 2007. Citeseer. 2007, pp. 1–16.
- [96] Claude E. Walston and Charles P. Felix. “A Method of Programming Measurement and Estimation”. In: *IBM Systems Journal* 16.1 (1977), pp. 54–73.
- [97] Barry W. Boehm. “Software Engineering Economics”. In: *IEEE Trans. Software Eng.* 10.1 (1984), pp. 4–21.
- [98] Phillip G. Armour. “Beware of counting LOC”. In: *Communications of the ACM* 47.3 (2004), pp. 21–24.
- [99] Martin P. Ward. “Language-Oriented Programming”. In: *Software — Concepts and Tools* 15.4 (1994).
- [100] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. “Program Transformation with Scoped Dynamic Rewrite Rules”. In: *Fundamenta Informaticae* 69.1-2 (2006), pp. 123–178.

- [101] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. “The mCRL2 Toolset for Analysing Concurrent Systems - Improvements in Expressivity and Usability”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*. Vol. 11428. Lecture Notes in Computer Science. Springer, 2019, pp. 21–39. DOI: 10.1007/978-3-030-17465-1_2.
- [102] Roberto Minelli, Andrea Mocci, and Michele Lanza. “I know what you did last summer: an investigation of how developers spend their time”. In: *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*. ACM, 2015, pp. 25–35.
- [103] Aron Zwaan. “Composable type system specification using heterogeneous scope graphs”. MA thesis. 2021.
- [104] Leonard G. C. Hamey and Shirley Goldrei. “Implementing a Domain-Specific Language Using Stratego/XT: An Experience Paper”. In: *Electronic Notes in Theoretical Computer Science* 203.2 (2008), pp. 37–51. DOI: 10.1016/j.entcs.2008.03.043.
- [105] Giovanna Broccia, Alessio Ferrari 0001, Maurice H. ter Beek, Walter Cazzola, Luca Favalli, and Francesco Bertolotti. “Evaluating a Language Workbench: from Working Memory Capacity to Comprehension to Acceptance”. In: *31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 2023, pp. 54–58. DOI: 10.1109/ICPC58990.2023.00017.
- [106] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. “Implementation of a Modelica compiler using JastAdd attribute grammars”. In: *Science of Computer Programming* 75.1-2 (2010), pp. 21–38. DOI: 10.1016/j.scico.2009.07.003.
- [107] Torbjörn Ekman and Görel Hedin. “The JastAdd extensible Java compiler”. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 2007, pp. 1–18. DOI: 10.1145/1297027.1297029.
- [108] Bas Basten, Jeroen van den Bos, Mark Hills 0001, Paul Klint, Arnold Lankamp, Bert Lisser, Atze van der Ploeg, Tijs van der Storm, and Jurgen J. Vinju. “Modular language implementation in Rascal - experience report”. In: *Science of Computer Programming* 114 (2015), pp. 7–19. DOI: 10.1016/j.scico.2015.11.003.
- [109] *Design of a domain-specific language for material flow analysis using Microsoft DSL Tools: An experience paper*. 2014.

- [110] Rutger van Beusekom, Jan Friso Groote, Paul F. Hoogendijk, Robert Howe, Wieger Wesselink, Rob Wieringa, and Tim A. C. Willemse. "Formalising the Dezyne Modelling Language in mCRL2". In: *Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18-20, 2017, Proceedings*. Vol. 10471. Lecture Notes in Computer Science. Springer, 2017, pp. 217–233. DOI: 10.1007/978-3-319-67113-0_14.
- [111] Ivan Kurtev, Mathijs Schuts, Jozef Hooman, and Dirk-Jan Swagerman. "Integrating Interface Modeling and Analysis in an Industrial Setting". In: *MODELSWARD*. SciTePress, 2017, pp. 345–352.
- [112] Ivo Raedts, Marija Petkovic, Yaroslav S. Usenko, Jan Martijn E. M. van der Werf, Jan Friso Groote, and Lou J. Somers. "Transformation of BPMN Models for Behaviour Analysis". In: *Modelling, Simulation, Verification and Validation of Enterprise Information Systems, Proceedings of the 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS-2007, In conjunction with*. INSTICC PRESS, 2007, pp. 126–137.
- [113] Jasper Denkers. "A longitudinal field study on creation and use of domain-specific languages in industry". In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 1152–1155. DOI: 10.1145/3338906.3341463.

Curriculum Vitae

Jasper Denkers

Born in 1995 in Rijnwoude, the Netherlands.

<https://www.jasperdenkers.nl>

2018 - 2024

Ph.D. in Computer Science

Delft University of Technology

Department of Software Technology, Programming Languages group

2011 - now

Self-Employed Software Engineer

DevelopTheWeb

2015 - 2018

M.Sc. in Computer Science

Delft University of Technology

Specialization: Software Technology

2015 - 2017

Full Stack Developer & Project Lead

Lunatech Labs

2012 - 2015

B.Sc. in Computer Science

Delft University of Technology

Minor: Finance

2006 - 2012

VWO diploma

Scala College in Alphen aan den Rijn

Specialization: Nature & Technology

List of Publications

- Jasper Denkers, Louis van Gool, and Eelco Visser. “Migrating custom DSL implementations to a language workbench (tool demo)”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018*. ACM, 2018, pp. 205–209. DOI: 10.1145/3276604.3276608
- Jasper Denkers. “A longitudinal field study on creation and use of domain-specific languages in industry”. In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 2019, pp. 1152–1155. DOI: 10.1145/3338906.3341463
- Jasper Denkers, Marvin Brunner, Louis van Gool, and Eelco Visser. “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*. Vol. 13085. Lecture Notes in Computer Science. Springer, 2021, pp. 423–442. DOI: 10.1007/978-3-030-92124-8_24
- Jasper Denkers, Marvin Brunner, Louis van Gool, Jurgen J. Vinju, Andy Zaidman, and Eelco Visser. “Taming complexity of industrial printing systems using a constraint-based DSL: An industrial experience report”. In: *Software: Practice and Experience* (2023). DOI: 10.1002/spe.3239
- Olav Bunte, Jasper Denkers, Louis van Gool, Jurgen J. Vinju, Eelco Visser, Tim Willemse, and Andy Zaidman. “OIL: an Industrial Case Study in Language Engineering with Spoofox”. In: *Software and Systems Modeling* (2024). DOI: 10.1007/s10270-024-01185-x

Titles in the IPA Dissertation Series since 2021

- D. Frumin.** *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2021-01
- A. Bentkamp.** *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VU. 2021-02
- P. Derakhshanfar.** *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03
- K. Aslam.** *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04
- W. Silva Torres.** *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05
- A. Fedotov.** *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01
- M.O. Mahmoud.** *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02
- M. Safari.** *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03
- M. Verano Merino.** *Engineering Language-Parametric End-User Programming Environments for DSLs.* Faculty of Mathematics and Computer Science, TU/e. 2022-04
- G.F.C. Dupont.** *Network Security Monitoring in Environments where Digital and Physical Safety are Critical.* Faculty of Mathematics and Computer Science, TU/e. 2022-05
- T.M. Soethout.** *Banking on Domain Knowledge for Faster Transactions.* Faculty of Mathematics and Computer Science, TU/e. 2022-06
- P. Vukmirović.** *Implementation of Higher-Order Superposition.* Faculty of Sciences, Department of Computer Science, VU. 2022-07
- J. Wagemaker.** *Extensions of (Concurrent) Kleene Algebra.* Faculty of Science, Mathematics and Computer Science, RU. 2022-08
- R. Janssen.** *Refinement and Partiality for Model-Based Testing.* Faculty of Science, Mathematics and Computer Science, RU. 2022-09
- M. Laveaux.** *Accelerated Verification of Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2022-10
- S. Kochanthara.** *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01
- L.M. Ochoa Venegas.** *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02
- N. Yang.** *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03
- J. Cao.** *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04
- K. Dokter.** *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05

- J. Smits.** *Strategic Language Workbench Improvements.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06
- A. Arslanagić.** *Minimal Structures for Program Analysis and Verification.* Faculty of Science and Engineering, RUG. 2023-07
- M.S. Bouwman.** *Supporting Railway Standardisation with Formal Verification.* Faculty of Mathematics and Computer Science, TU/e. 2023-08
- S.A.M. Lathouwers.** *Exploring Annotations for Deductive Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09
- J.H. Stoel.** *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software.* Faculty of Mathematics and Computer Science, TU/e. 2023-10
- D.M. Groenewegen.** *WebDSL: Linguistic Abstractions for Web Programming.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11
- D.R. do Vale.** *On Semantical Methods for Higher-Order Complexity Analysis.* Faculty of Science, Mathematics and Computer Science, RU. 2024-01
- M.J.G. Olsthoorn.** *More Effective Test Case Generation with Multiple Tribes of AI.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02
- B. van den Heuvel.** *Correctly Communicating Software: Distributed, Asynchronous, and Beyond.* Faculty of Science and Engineering, RUG. 2024-03
- H.A. Hiep.** *New Foundations for Separation Logic.* Faculty of Mathematics and Natural Sciences, UL. 2024-04
- C.E. Brandt.** *Test Amplification For and With Developers.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05
- J.I. Hejderup.** *Fine-Grained Analysis of Software Supply Chains.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06
- J. Jacobs.** *Guarantees by construction.* Faculty of Science, Mathematics and Computer Science, RU. 2024-07
- O. Bunte.** *Cracking OIL: A Formal Perspective on an Industrial DSL for Modelling Control Software.* Faculty of Mathematics and Computer Science, TU/e. 2024-08
- R.J.A. Erkens.** *Automaton-based Techniques for Optimized Term Rewriting.* Faculty of Mathematics and Computer Science, TU/e. 2024-09
- J.J.M. Martens.** *The Complexity of Bisimilarity by Partition Refinement.* Faculty of Mathematics and Computer Science, TU/e. 2024-10
- L.J. Edixhoven.** *Expressive Specification and Verification of Choreographies.* Faculty of Science, OU. 2024-11
- J.W.N. Paulus.** *On the Expressivity of Typed Concurrent Calculi.* Faculty of Science and Engineering, RUG. 2024-12
- J. Denkers.** *Domain-Specific Languages for Digital Printing Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-13



www.jasperdenkers.nl

