EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Towards Model Driven Engineering for Carriage Motion

van der Pol, A.F. (Bram)

Award date:
2022

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Department of Mathematics and Computer Science
Software Engineering and Technology Research Group

Towards Model Driven Engineering for Carriage Motion

Master Thesis

Abraham Folkert van der Pol
0780042

Supervisors:
Loek Cleophas (TU/e)
Joost van Pinxten (CPP)
Jeroen Lind (CPP)

Committee Members:
Loek Cleophas (TU/e)
Joost van Pinxten (CPP)
Tim Willemse (TU/e)

Eindhoven, September 2022

Canon
CANON PRODUCTION PRINTING

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

Abstract

This Master Thesis written at Canon Production Printing investigates the feasibility of using DSL models to generate printer carriage motion profiles that can be executed by existing embedded software. This was demonstrated for the Colorado Wide Format printer by implementing a prototype library that implements DSL concepts in C++, which is integrated in the UML-RT-based embedded software for the Colorado printer. While the results only directly target a specific printer, the approach is applicable to other Wide Format printers. In general, the approach could be used in other disciplines where existing embedded software is in need of higher-level modeling for part of the existing functionality.

Preface

I would like to extend my gratitude to everyone who helped me in the process of writing this thesis. I am thankful to Canon Production Printing for allowing me to work on this graduation assignment inside the company. In particular I want to thank Joost and Jeroen from CPP for providing guidance and valuable feedback on my progress throughout my internship, both inside and outside our weekly meetings. I also want to thank Loek for mentoring me from the TU/e side of things, for giving feedback on the numerous iterations of this thesis and for his support in general. Finally I would like to mention Michel Chaudron whose comments on my progress often helped me see things in a new light.

Contents

Contents vii

1 Introduction **1**
1.1 Context . . . 1
1.2 Motivation . . . 2
1.3 Problem Statement . . . 2
1.4 Research Questions . . . 3
1.5 Thesis Outline . . . 4

2 Background **5**
2.1 Related Work . . . 5
2.2 Large Format Graphics . . . 6
2.2.1 Layout and Movement . . . 6
2.2.2 Digital Image to Ink Jetting . . . 6
2.2.3 Curing . . . 7
2.2.4 Carriage Control . . . 7
2.3 Summary . . . 8

3 Existing Situation **9**
3.1 Embedded Software - RTist . . . 9
3.1.1 RTist Concepts . . . 10
3.1.2 Architecture . . . 10
3.1.3 Module Interactions . . . 11
3.1.4 SIL Simulator . . . 13
3.2 DSLs - MPS . . . 13
3.2.1 Print System Layout DSL . . . 14
3.2.2 Scanning Productivity DSL . . . 16
3.3 Summary . . . 18

4 Approach **19**
4.1 Options for Determining Feasibility . . . 19
4.2 High Level Connectivity Options . . . 19
4.2.1 Generate an RTist Model . . . 20
4.2.2 Connect at DSL Level Through ESME . . . 20
4.2.3 Generate TargetRTS Compatible C++ Code . . . 20
4.2.4 Generate Stand-alone C++ Library . . . 21
4.2.5 Overview . . . 21
4.3 Qualitative Evaluation . . . 23
4.4 Chosen Approach . . . 23
4.5 Alternative Approach . . . 23
4.6 Summary . . . 24

5 Relating CM and ESW **25**
5.1 Situations Versus Swaths . . . 25
5.2 Trajectories Versus Swaths . . . 26
5.3 Alignables . . . 27
5.4 Parametrized Alignables . . . 27
5.5 Synchronizing End Situations . . . 27
5.6 Component Order . . . 28
5.7 Summary . . . 28

6 Prototype **29**
6.1 Mapping DSL Concepts to Prototype . . . 29
6.2 Integrating Schedules in ESW . . . 30
6.3 Interfaces . . . 31
6.4 Summary . . . 32

7 Evaluation **33**
7.1 Verification Plan . . . 33
7.2 Verification Options . . . 33
7.2.1 Unit Tests . . . 34
7.2.2 Simulator . . . 34
7.2.3 Embedded Software Logging Data . . . 34
7.2.4 Physical Printer Engine . . . 35
7.3 SIL Simulator Suitability Justification . . . 35
7.4 Experimental Results . . . 35
7.4.1 Default Print Mode . . . 35
7.4.2 Pin Cure Print Mode . . . 36
7.4.3 Replicating Original Pin Cure Alignments . . . 38
7.5 Limitations . . . 39
7.5.1 Delayed Start . . . 39
7.5.2 Light Timing . . . 39
7.5.3 Evaluation Scope . . . 39
7.5.4 Manual Implementation . . . 40
7.6 Summary . . . 40

8 Colorado Carriage Motion DSL Outline **41**

9 Conclusions **43**

Bibliography **47**

Glossary **49**

Appendix A **51**

Chapter 1

Introduction

This thesis is the result of my graduation internship at Canon Production Printing. It lays out the steps towards bridging the gap between domain-specific models for the behaviour of carriage motion, and their execution in the embedded software.

1.1 Context

Canon Production Printing (CPP) develops digital inkjet printers for professional applications. While CPP makes different types of printers, this project focuses on the Large Format Graphics printers. These printers are also known as roll-to-roll printers, as the material on which is printed (the print medium) is fed from a roll, passes through the printer, and after printing is rolled back onto a roll. Current generation Large Format Graphics printers use two carriages that move back and forth over the print medium in a scanning motion. One carriage deposits ink on the medium, while the second carriage is used to cure (dry) the ink. For most of this thesis a specific printer engine is used: the Colorado printer family [12], depicted in Figure 1.1. Figure 1.2 shows the carriages which move horizontally over the print material. From this point of view, this material passes through the printer from top to bottom, perpendicular to the carriage movement.

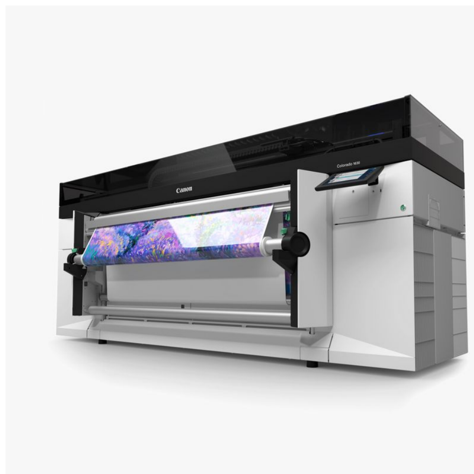

Figure 1.1: The Colorado printer, side view.

Figure 1.2: The Colorado printer, top view.

To ensure high quality prints, the movement of these carriages needs to be precisely controlled. Apart from high quality prints, another important attribute of a printer is productivity. Printers have different print modes, so that the operator can select the right quality and productivity

trade-off for the intended application. Depending on the desired end result, different motion strategies can be selected, which influences how the print and cure carriages align with each other. These motion strategies are manually encoded in embedded software, which becomes complex for a multitude of print modes and hardware configurations. This complexity makes it difficult to maintain the software and to make sure each print mode operates as intended. To keep the software maintainable, an alternative approach to motion strategy encoding is needed.

1.2 Motivation

As is a trend in many domains [7], CPP printers rely on increasingly complex embedded software for their operation. To combat the issue of rising complexity, CPP is investing in Model-Based Development using Domain Specific Languages (DSLs). These DSLs are developed using JetBrains Meta Programming System (MPS) [5]. The idea is to capture the important domain details of a subtask in the printer development process in a language so that a domain expert can focus on modeling, without getting bogged down in details or intricacies of more generic modeling tools.

One such language is dedicated to expressing the movement of multiple carriages in a carriage-based print system. Using this language, models of movement profiles for different print modes can be expressed in an intuitive way. These models are currently being used for making design choices, but are not involved in later stages of the design process. In a sense this is a waste; by not reusing models, the same behaviour has to be encoded by humans multiple times, costing time, effort and increasing likelihood of mismatches between modeled behaviour and the related implementation. A depiction of the envisioned way of working based on DSL models is given in Figure 1.3.

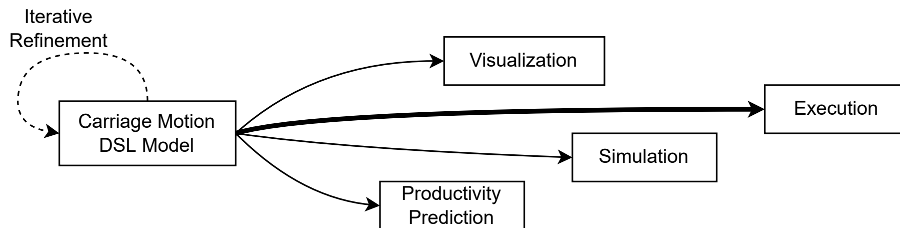

Figure 1.3: Envisioned way of working based on DSL models, with bold arrow highlighting the focus of this thesis.

Domain specific models expressing a motion strategy are easier to specify, analyse and verify than their embedded software counterparts. Such models are more concise and are more limited in scope compared to implementations of these models in generic programming languages. Furthermore, DSL models exist as separate modules, while generic implementations often rely on boilerplate code unrelated to the modeled behaviour. If these DSL models could somehow be integrated in the embedded software, much of the complexity around motion profiles moves from the embedded software to the DSL. This greatly improves maintainability and simplifies verification of the embedded software. Motion profiles could have a single source of truth that is theoretically analysed and verified, which is also used in the execution of the motion profile on an actual printer.

1.3 Problem Statement

As motivated in the previous paragraph there are many advantages to shifting complexity from embedded software to DSL models. The embedded software however, is written in RTist. RTist is based on UML-RT, a real time derivative of UML that uses state machines and active classes. This UML-RT software generates to C++. While the DSL is not in C++ format, generating

some C++ from a DSL model is not difficult. Generating C++ that integrates with the existing RTist software is a different question entirely, and is much more complex. Besides that, there are conceptual differences between specification of motion strategies between the DSL and the embedded software, complicating matters further. So, while the advantages of using a single source of truth and moving complexity to a more easily analysable domain are clear, it is not so clear if it is feasible to transform the models for use in the existing embedded software. The main problem that is considered in this thesis is to find out whether this transformation is feasible or not.

1.4 Research Questions

The first and main research question follows from the problem statement. For the remaining research questions a brief motivation for including the research question is given. The main research question is defined as follows:

> **RQ1**: Is it possible to use Model Driven Engineering to transform carriage motion DSL models to RTist-based embedded software?

There are several challenges that come with this main question, which include conceptual differences, semantic gaps, generating C++, and dealing with UML-RT models. However, we need to find the main challenges among these. This also allows us to analyse where the blocking issue occurs in case the main question result turns out to be infeasible. Hence, the second question:

> **RQ2**: What are the main challenges in translating models expressed in CPP's Carriage Motion DSL developed in JetBrains MPS to RTist models?

The next question can be seen as an extension to RQ2, as the question posed is part of the challenges that need tackling. However, DSL changes are a specific topic that deserve attention. At the beginning of the assignment it was expected that some modifications to the existing DSL would be necessary. It could be that more information needs to be added to the DSL to facilitate the envisioned transformation, or perhaps the DSL needs constraining in order for transformation to be possible.

> **RQ3**: What is the semantic gap between the existing DSL specification and the RTist-based embedded software?

Because it was unclear at the beginning of this thesis how the DSL and embedded domain are linked, and there is a significant gap, it is expected that a number of tools could be used. Especially generating code from the DSL models was thought to be something that could be done by different tools. This last part however can be done using built in MPS functionality, namely the generator aspect of MPS. This question may be trivial, but it may be interesting for the reader to learn that generating code from a DSL can have a relatively straight forward solution. So, RQ4:

> **RQ4**: Which tooling can be used to translate these JetBrains MPS models to RTist models?

The final question can only be answered definitively if CPP decides to move to the Model Driven Engineering approach that is the topic of this research. Is moving to model-based engineering for carriage motion an improvement over the current way of working?

> **RQ5**: What is the efficacy of using Model Driven Engineering for carriage motion execution (in terms of reduced development time or improved functional correctness guarantees)?

1.5 Thesis Outline

The remainder of this thesis is structured as follows: first, some background information is given regarding printer terminology and related works, followed by a description of DSLs and the embedded software as they were at the start of this assignment. Next, an overview of possible approaches to tackle the main research question is given, including a rationale for picking a certain one, followed by a more in-depth dive on mapping DSL concepts to the embedded software. Then a prototype implementation is described and experimentally evaluated before ending with concluding remarks.

Chapter 2

Background

After treating related work, this chapter gives some background about Large Format Printers and the terminology that is used in this context.

2.1 Related Work

Perhaps the most relevant previous work is by Schindler et al. [13]. This book chapter gives an insight into how CPP uses MPS-based models to aid in the engineering practice, and introduces several domain specific languages used in various contexts. While the different DSLs for the range of contexts are interesting to learn about, pages [28-33] are particularly relevant to this thesis. This excerpt talks about Virtual Printer Configuration, and two DSLs which will be introduced in more detail in Section 3.2 in this thesis. One DSL is geared towards printer layout, while another is used to specify carriage motion strategies. While initially these DSLs were designed with virtual printers in mind, here we will reuse them to facilitate execution.

In his PDeng thesis, Nikeshin [10] introduces Embedded Software Modelling Environment (ESME). ESME is an MPS-based tool that is capable of loading RTist models, converting them to MPS models, and generating real time C++ code from these MPS models. The intent of ESME is to replace the RTist IDE, and allow model users to leverage the power of MPS when writing models. ESME was considered as a stepping stone between carriage motion DSL models and the embedded software, which is described in Section 4.2.2. While ESME is at proof-of-concept stage, it is an interesting take on transforming RTist models to the MPS domain.

Maheshwari et al. [8] discusses generating model-based artifacts for legacy systems. They introduce the Diamond Process Model, a systematic approach to generating model-based artifacts from legacy systems. The focus is on reverse engineering existing systems, and capturing the information in these systems in requirement sets. From these requirement sets forward engineering is then applied to generate models satisfying the found requirements. The authors identify three reasons for doing so. Firstly, to integrate legacy systems with new systems that were engineered using Model-Based Systems Engineering (MBSE). Secondly, to upgrade or to provide new capabilities to legacy systems. Lastly, to capture the design of legacy systems in models for use in future systems to be designed using MBSE. Especially the first and second reason have close resemblance to the topic of this thesis; existing DSL models that can be seen as new MBSE-based system components are integrated with a legacy embedded software system, which provides new capabilities to the embedded software.

A framework for integrating model-based artifacts in existing embedded software is introduced by Ohno et al. [11]. In particular the framework is aimed at connecting the interfaces between artifacts generated from control system models, and the embedded software that uses these artifacts. The work done in this paper has similarities to the subject of this thesis. However, the framework expects existing artifacts in the language of the embedded software, and merely automates the connection of relatively simple interfaces between these artifacts and the embedded software. In

our case we first need to bridge a semantic gap between DSL models and the embedded software before any artifacts can be generated.

In Czech et al. [1] a mapping study on best practices for domain-specific modeling (DSM) is laid out. From 21 selected papers they compiled a list of 192 best practices related to DSM. While two of these practices, reusing languages and providing integrability, touch upon the practices relevant to our research, deriving domain specific models from, or integrating with existing software was not mentioned.

2.2 Large Format Graphics

This section goes more in depth in printer terminology and introduces the relevant details of the Colorado printer family [12], which is the specific printer engine for which the embedded software is considered in this thesis. Most terms are explained in the text, but for reference one may refer to the list of definitions at the back of this document.

2.2.1 Layout and Movement

We will consider the printer as seen from the top, with the wider side in front of us, as shown in Figure 2.1. The carriage movement is therefore a side to side movement, while the print medium moves from top to bottom. The latter is called a media step. The movement of the print medium is perpendicular to the movement of the carriages, and is handled outside carriage motion. When planning carriage motion it is relevant to take the duration of a media step into account, as jetting a new line can only be started after the print medium movement is completed. The print medium is not allowed to move during jetting. Otherwise the media step is not relevant to carriage motion. The distance between the print heads and the print medium (the carriage height) is adjustable, but likewise falls outside the scope of carriage motion. One might also say the carriage motion problem is a one dimensional problem, having one degree of freedom.

Figure 2.1: Colorado top view, high-lighting carriages and schematic dotted track.

2.2.2 Digital Image to Ink Jetting

A digital image consists of pixels. The Colorado printer prints by depositing ink droplets, dots, on the print medium. Note that one dot does not correspond to one pixel. The pixels of an image therefore need to be translated to dots to be printed. The dots are placed by jetting ink from nozzles which are contained in the print heads. Each print head is equipped with hundreds of nozzles. A typical image will have a height that is larger than the height of the print heads. So, the dotted image needs to be cut in slices that are printed sequentially. The printer prints a slice, then the print medium is moved a small amount so that the subsequent slice can be jetted at the right distance from the previous slice. This process is repeated until the entire image is jetted.

The printing of a slice is also known as the execution of a swath. Typically the media step is smaller than the height of the print heads. The print heads therefore pass over each piece of the print medium where jetting needs to take place multiple times. A print mode where the media step is a quarter of the print head height causes each piece of the print medium to be passed by the print heads four times. A graphical depiction of such a four-pass print mode is given in Figure 2.2.

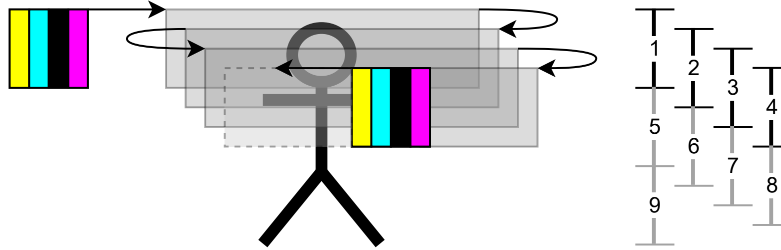

Figure 2.2: Schematic depiction of multi-pass printing with a four pass print mode, showing the first four swaths*.

2.2.3 Curing

The ink that is used in the Colorado family is an UV-gel-based ink. This ink needs to be cured by UV light before the print medium exits the printer. This curing is a multi step process; a combination of light intensity, cumulative light exposure, and the timing of this exposure influences the end result of the image that is printed on the print medium. Depending on the time between jetting ink on the print medium and curing that ink, the surface finish will appear matte or more glossy. If the time between jetting and curing is large, the ink droplets can flow out and become much flatter resulting in a glossy finish. If the ink is cured soon after it was jetted, the deposited ink droplets have little time to flow out, causing the droplets to retain a dome-like shape. This yields a matte finish. The ink droplets are “pinned” in place as it were, which is why this type of curing is called pin curing.

In the Colorado printer there are no cure lights on the print carriage. It is not possible to achieve a matte result using just the cure carriage, as there is too much time between the ink being jetted and the ink passing under the cure carriage. To solve this, the Colorado has mirrors on the print carriage that are positioned in such a way that they reflect light from the cure carriage onto the print medium. In order for light from the cure carriage to shine on the mirrors, the two carriages need to be aligned while pin curing is in progress. A schematic representation of the Colorado print- and cure carriages, including pin cure alignment, is shown in Figure 2.3. In this figure both carriages are travelling from right to left, while being aligned on the left pin cure mirror. When the carriages travel in the opposite direction alignment would take place between the pin cure light and the right mirror for this particular print mode. Several pin cure modes exist, each having a different sequence of pin cure alignments.

2.2.4 Carriage Control

To make sure that ink is jetted at the intended location, there are several requirements on carriage movement. During printing, the print carriage moves over the print medium. Because there is some distance between the nozzles and the print medium, the ink leaving the nozzles hits the print medium at an angle. This angle changes with carriage speed. To achieve an even surface finish, it is important that the print carriage remains at constant speed while ink is being jetted.

*In reality the left and right side of the depicted swaths would be identical for all swaths. Here each subsequent swath is offset to better show the distinction between them.

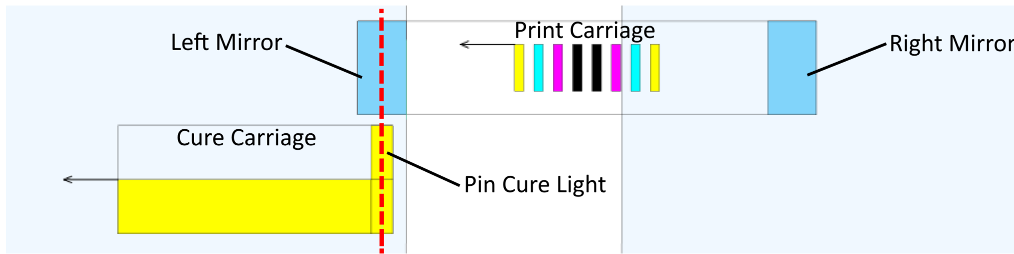

Figure 2.3: Colorado carriage alignment during pin curing. The center of the mirror on the print carriage is aligned with the center of the pin cure light on the cure carriage (dashed line).

Besides the need for constant speed, there is also a maximum velocity that is allowed by the mechanical system. The same goes for the first and second derivative of velocity, the acceleration, and jerk. At no point in time can the maximum velocity, acceleration or jerk be exceeded. While controlling the carriage to prevent these values from being exceeded is out of scope, the fact these limits are in place has an effect on the behaviour of carriage motion. These physical limits govern the minimum time a turn can take, or the minimum distance that is needed for a carriage to go from stationary to the constant speed required during jetting.

2.3 Summary

In this chapter we first looked at literature related to this thesis. While some previous work on integrating model-based artifacts in existing software was found, there seems to be little literature on the topic. Next we went into some details pertaining to the inner workings of Large Format Graphics printer engines. In particular we considered the carriage control and curing characteristics of the Colorado family printer engines. In the next chapter we will examine the existing DSLs and embedded software for the Colorado printers used at CPP.

Chapter 3

Existing Situation

This chapter introduces the existing situation in the embedded software and DSLs at the start of this assignment. First we take a look at the platform on which the existing embedded software is built, and consider parts of the architecture and the software components relevant for carriage motion control. Next we take a look at the platform for the DSLs and describe relevant existing DSLs.

3.1 Embedded Software - RTist

CPP uses the RTist modeling tool and RTist Real-Time Engine for their embedded software [3]. The RTist modeling tool is an Eclipse based IDE supporting UML-RT with a code generation back end that generates C++ to be run in the RTist runtime environment. By writing UML-RT models, developers can focus on modeling behaviour without having to deal with threads or C++ implementation details. This is done by reasoning in terms of state machines and message passing. State machines are annotated with C++ code, which is executed when transitioning to a different state. Figure 3.1 shows an example of such a state machine. These annotated state machines can then be generated to C++ code. This generated C++ code uses the RT Services Library runtime framework to provide runtime implementations of the real-time concepts contained in RTist models. The RT Services Library is also referred to as TargetRTS (Target Run Time System). Figure 3.2 gives an overview of the platform and corresponding artifacts introduced in this section.

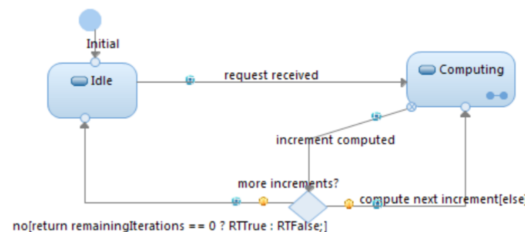

Figure 3.1: Example state machine in RTist UML-RT model

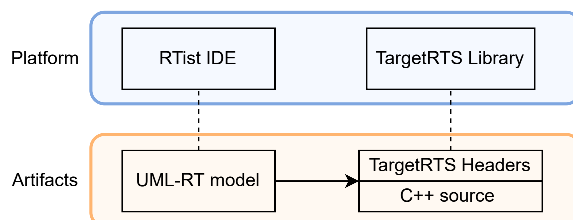

Figure 3.2: RTist platform and artifacts

RTist is developed by HCL, and is derived from Rational Software Architect RealTime Edition (RSARTE) which used to be maintained by IBM. RSARTE in turn is an evolution of Rational RoseRT which has been around since the late 1990s.

3.1.1 RTist Concepts

The main building blocks in an UML-RT model are *capsules*. Capsules contain the modules structure, attributes and methods, and a single state machine. The latter contains the behaviour of the module, and is also why capsules are sometimes referred to as active classes; each state machine inside a capsule runs independently. With a multitude of capsules these state machines run concurrently, similar to parallel threads. Capsules can contain other capsules forming a composite structure.

Capsules are able to communicate by sending messages through *ports*. For each port a *protocol* is defined, and only ports that use the same protocol can be connected. When a capsule receives a message, this can trigger a state transition in its state machine. A state transition can have a guard, so that the state machine can control whether a transition fires. This is the main mechanism through which the behaviour of the system is modelled. State machine transitions are annotated with C++ code. When a state machine transition fires, the accompanying C++ code is executed. To facilitate the execution of the software, RTist contains TargetRTS; the Target Run Time System. The TargetRTS provides an implementation for the environment in which the capsules can be executed, similar to real time operating systems. For more details on RTist concepts one may refer to [9].

3.1.2 Architecture

The embedded software consists of many capsules that are able to communicate and synchronize through channels, but otherwise run in parallel. The separation of certain processes was clearly a target for the architecture. The result is that the capsules responsible for the carriage motion related functionality have little overlap with other functionality. This description therefore focuses on the capsules and architectural elements that are relevant for controlling the carriage movement.

A concise overview of the relevant modules and their responsibilities is given, followed by a more in depth look at each capsule and the interactions between them. A diagram depicting the connections between modules is given in Figure 3.3.

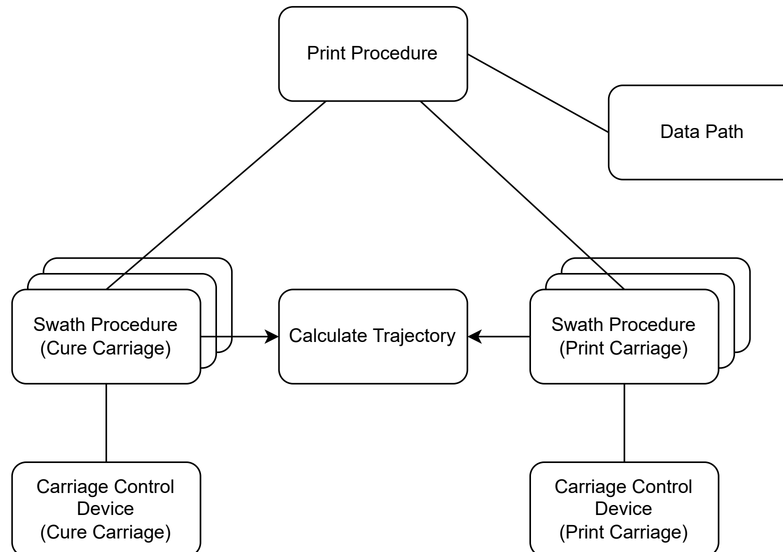

Figure 3.3: Connections between relevant RTist modules.

- **Data Path**: Decodes the image, cuts the image in slices that are to be jetted, and determines the boundary positions for each slice (that is to be swathed). The actual jetting of the image is handled by a separate capsule.

- **Print Procedure**: Executes a print job. Using the Data Path boundaries as input, this procedure schedules and synchronizes the time and position of movements of the print carriage, the cure carriage, and the media handling (i.e. the movement of the print medium between swaths). This cycle repeats for each required swath until the print job is completed.
- **Swath Procedure**: Takes positional as well as start time info from the Print Procedure for a single swath, calculates the required carriage motion profile, and negotiates execution timing with the Print Procedure. Has a scheduling phase and an execution phase. For each carriage, multiple instances of swath procedures are running. During normal printing one of these procedures controls the carriage control device executing the current swath and one is planning the next swath.
- **Calculate Trajectory**: Function that takes positional as well as start time info for a single swath and calculates the required carriage motion and resulting timing profile in a detailed 12 step sequence. This originated as a Matlab script, but is converted to C code for use in the embedded software. This conversion falls outside the scope of this thesis.
- **Carriage Control Device**: Takes the 12 step sequence from calculate trajectory and uses a high-frequency control loop to execute this sequence by driving the carriage motors using carriage position encoder feedback.

3.1.3 Module Interactions

The main interactions we focus on are those that occur while steady state printing. This is the repeating pattern of executing swaths until the entire image is jetted on the print medium. This process is explained in words, but is also captured in a UML sequence diagram in Figure 3.4.

The process starts with a swath request from the Data Path arriving at the Print Procedure. This swath request includes absolute start and end positions for the print carriage. Next, the Print Procedure queries an idle Cure Swath Procedure instance for the cure swath that is required, which depends on the print swath request and the selected curing mode. The Cure Swath Procedure relays the required swath begin and end positions, the cure swath request, to the Print Procedure. Once the required swath information is known, an initial trajectory is calculated for both carriages so that the earliest time each carriage can start executing the requested trajectory is known. This is done by again querying the selected Cure Swath Procedure instance, and an idle instance of a Print Swath Procedure.

In nearly all cases the earliest start time for a requested trajectory will differ between the carriages. These two trajectories however have to start at the same moment in order for the carriage movement to remain in sync. Therefore the start times of one of the trajectories is delayed to synchronize the carriage trajectories. This involves another call to the carriage trajectory calculation; in some cases the trajectory remains largely the same, only with a later start time, but in other cases a later requested start time causes a need for additional prepended trajectory sections. This happens for example when the initial trajectory allowed for continuous carriage movement, while the delayed start causes the carriage to require a stop. A new trajectory for the second carriage in turn may cause the original carriage to no longer be able to execute the planned trajectory for similar reasons. This is why the synchronization process may take several iterations to converge. Eventually this process always converges. When a trajectory is planned that requires a carriage to stop, this trajectory can always be delayed arbitrarily long while remaining executable.

Once a trajectory with synchronized start times is found for both carriages, the trajectories are queued for execution. Next, the Print Procedure waits until a current swath and subsequent media step are finished. The trajectories that were just calculated are then executed, at which moment the Print Procedure is ready to start planning the next swaths. Once the trajectories have finished executing, the Cure- and Print Swath Procedure instancess become idle again.

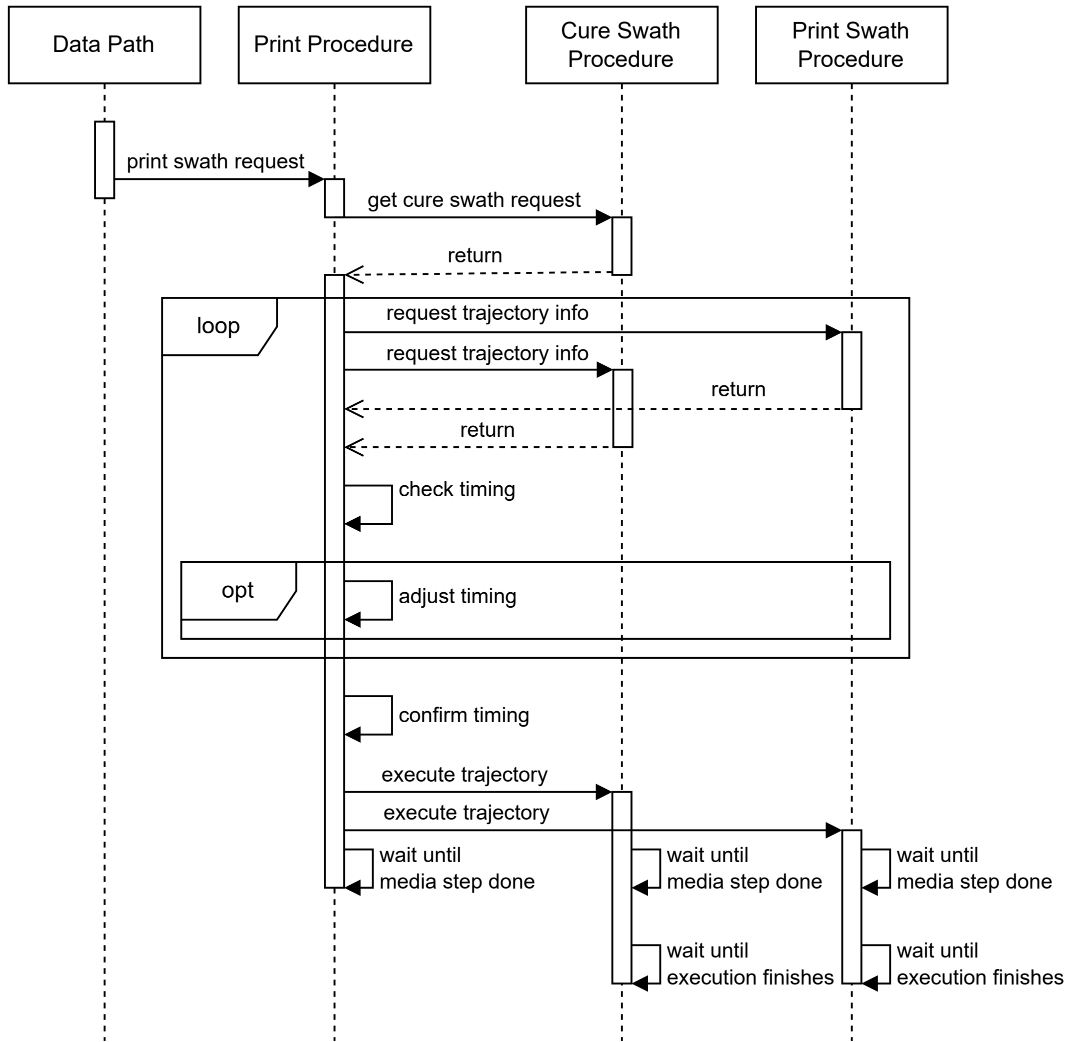

Figure 3.4: Sequence diagram depicting a single iteration of the steady state printing loop*.

Calculate Trajectory

The trajectory calculator takes a swath begin position, swath end position, and requested start time, and uses these inputs to calculate a carriage trajectory to facilitate that swath. Depending on the current position and speed of the carriage it may be the case that the carriage just has to accelerate in one direction before the swath can begin. If there is too little space to accelerate to the desired swath speed, it may be the case that the carriage first has to be moved in the direction opposite to the swath direction. This way enough distance is available for the run-up to reach the desired swath speed before the swath begin position.

To handle these different scenarios, the trajectory that is calculated is divided in a number of sections, shown in Figure 3.5. Some of these sections are always used, and some sections are optional and depend on the movements that need to be prepended to the swath. For each of these sections the begin and end times are calculated, which are used during the synchronization of swath start times for the different carriages. Once an agreement is reached, the calculated trajectory is given to the Carriage Control Device high-frequency control loop which executes the

*Please note this diagram is a simplification of reality, and does not take the Action Control Manager (ACM) into account. The ACM handles the plumbing between the different actors and allocates free Swath Procedure instances to the Print Procedure.

trajectory by driving the actual carriage motor. The carriage control loop is out of scope for the purposes of this thesis.

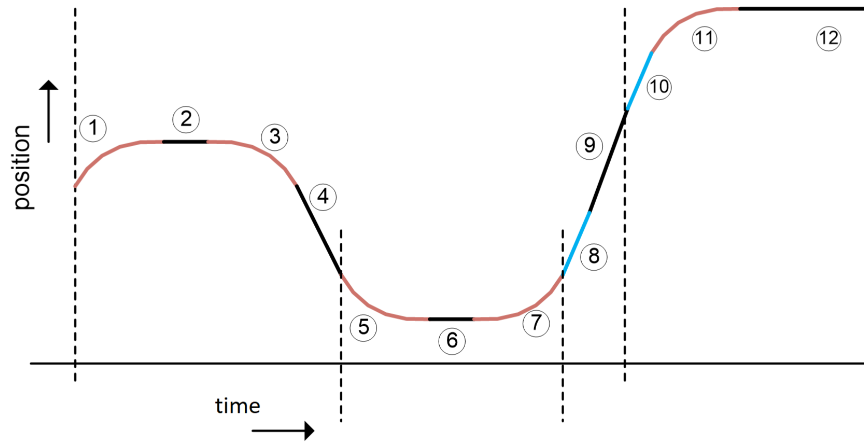

Figure 3.5: Calculate Trajectory sections. Section 9 is the trajectory section corresponding to a swath. Lower numbered sections may be prepended to a swath when necessary, while higher numbers make sure the carriage stops in case no subsequent swaths are queued.

3.1.4 SIL Simulator

The Software in the Loop (SIL) simulator is a simulated version of the printer engine - it emulates the behaviour of a physical printer engine in software. The simulator simulates the physical devices in the printer engine, as well as the control loops and the environment of the printer engine. This environment is capable of running the embedded software. The IO between the embedded software and the components of the printer engine is emulated. This is done by alternating the running of the embedded software, and halting the embedded software to handle outstanding IO requests. SIL allows running the embedded software without the need for a physical printer engine.

3.2 DSLs - MPS

At CPP, JetBrains Meta Programming System (MPS) is used for DSL development [5]. Languages for modeling different aspects of printers and for different types of printers are used or being developed. Schindler et al. [13] introduces some of these languages, including the languages that are relevant here. Because several concepts in these languages play an important role later in this thesis, these concepts are shown here in some more detail.

For this thesis, the main language of interest is the Scanning Productivity (SP) language, which [13] refers to as the Carriage Motion language. Models in this language describe the movement and synchronization of print carriages in carriage-based print systems. The naming also gives a hint as to what the language is used for - determining the productivity of different scanning motion strategies. This language makes use of the Print System Layout (PSL) language which describes the components in a print system and where these components are placed. For both these languages, models are written using a projectional textual editor. A graphical interface is available that can be used to visualize print system layouts and motion strategies, however it is not possible to modify models in this view. The latter is helpful in understanding models in a more intuitive sense.

In the following sections a more in-depth overview of the Scanning Productivity and Print System Layout languages is given, aided by both textual and graphical model representations.

3.2.1 Print System Layout DSL

Printer components are expressed in a DSL named Print System Layout (PSL). This language describes the layout of carriages in a print system in terms of components placed in 2D space as seen from above, similar to the floor plan of a building. Components can be composite, and multiple instances of a component may be used in a layout. Figure 3.6 shows part of a PSL model, in this case the print carriage definition. This carriage consists of a single print head with four ink colours. Figure 3.7 shows a textual representation of the same part of this model. While only a single carriage definition is shown here, a PSL model can contain any number of carriages, though the models shown in this thesis will usually have two carriages.

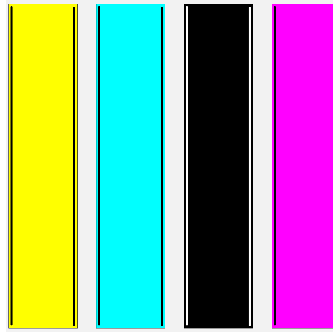

Figure 3.6: PSL DSL print carriage model - graphic representation.

```
[ Constants: ]

[ val slotWidth = 7 ;

[ Primitives: ]

staggered nozzle plate nozzlePlate 500 nozzles at 500 npi
  Stagger width: 5 mm

nozzle group PrintHead : rectangular
  placed nozzlePlate at (x, y)= (0 mm, 0 mm)
  << Custom Named Points >>
[ Printing: ]

print carriage PrintingCarriage @ 10 kHz

  Ink Channels:
   | Name | Pitch |
   | Y    | 10    |
   | C    | 10    |
   | M    | 10    |
   | K    | 10    |

  Drop sizes ( 1 ):
   | Volume [pl] | Diameter [μm] |
   | 1           | 10            |

  Print Heads:
    head Y1 ■ : PrintHead -> top left at (x, y) = (0 * slotWidth, 0) using Y
    head C1 ■ : PrintHead -> top left at (x, y) = (1 * slotWidth, 0) using C
    head M1 ■ : PrintHead -> top left at (x, y) = (3 * slotWidth, 0) using M
    head K1 ■ : PrintHead -> top left at (x, y) = (2 * slotWidth, 0) using K

  Extensions:
    << ... >>
```

Figure 3.7: PSL DSL print carriage model - textual representation of the same model depicted graphically in Figure 3.6.

3.2.2 Scanning Productivity DSL

Scanning Productivity (SP) models represent motion strategies by specifying a sequence of states, called Situations. A Situation describes carriage alignments in terms of some carriage component, and the position of that component in relation to some reference position. A reference position can be an edge of the print medium, or a printer component on a different carriage. Each SP model refers to a certain Print System Layout (PSL) model, as a motion strategy is defined in terms of components from a certain PSL model. A SP DSL model is also known as a Repeated Motion model.

Figure 3.8 and 3.9 contain a graphical and textual representation respectively of a Scanning Productivity model for a simple motion strategy. This simple motion strategy refers to the same PSL model from Section 3.2.1. The print carriage that was shown in Figure 3.6 can be seen in Figure 3.8 as well. The cure carriage which is defined in the same layout consists of a single curing light in this case, represented by a square containing a light bulb. The arrows next to the squares visualize the direction of motion. A saw tooth line separates the different Situations in this motion strategy.

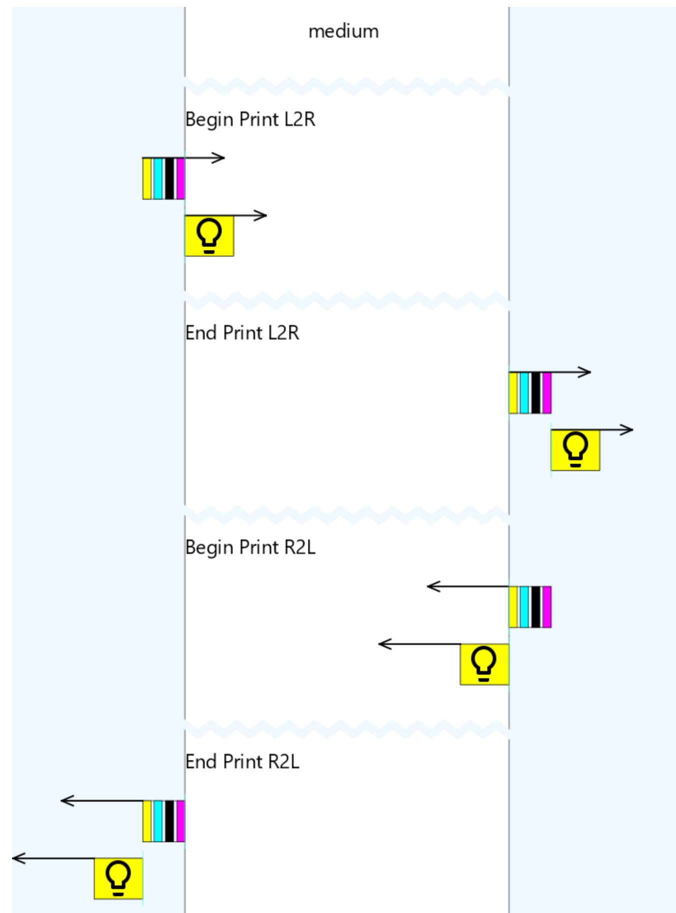

Figure 3.8: SP DSL Repeated Motion model - graphic representation.

DSL Concepts

Next we will look at some concepts of the SP DSL that will become important later on, as these concepts play a role in conceptual analysis of the DSLs and the embedded software in Chapter 5. Several of these concepts are also used in the prototype library that will be treated in Chapter 6.

```
repeated motion ExampleMotion<example_layout_spec>

  carriages: PrintCarriage CureCarriage
  medium: ResultImage

  situations:
    Begin Print L2R {
      (PrintCarriage) { right of ChannelM is aligned with left of ResultImage @ 1000 mm/s }
      (CureCarriage ) { left of Cure is aligned with right of ChannelM       @ 1000 mm/s }
    }
    End Print L2R {
      (PrintCarriage) { left of ChannelY is aligned with right of ResultImage @ 1000 mm/s }
      (CureCarriage ) { left of Cure is aligned with right of ChannelM       @ 1000 mm/s }
    }
    Begin Print R2L {
      (PrintCarriage) { left of ChannelY is aligned with right of ResultImage @ -1000 mm/s }
      (CureCarriage ) { right of Cure is aligned with left of ChannelY       @ -1000 mm/s }
    }
    End Print R2L {
      (PrintCarriage) { right of ChannelM is aligned with left of ResultImage @ -1000 mm/s }
      (CureCarriage ) { right of Cure is aligned with left of ChannelY       @ -1000 mm/s }
    }

  possible begin situations:
    << ... >>

  media steps:
    media step from End Print R2L to Begin Print L2R
    media step from End Print L2R to Begin Print R2L

  trajectories:
    constant speed for PrintCarriage : Begin Print L2R -> End Print L2R
    constant speed for PrintCarriage : Begin Print R2L -> End Print R2L
    constant speed for CureCarriage : Begin Print L2R -> End Print L2R
    constant speed for CureCarriage : Begin Print R2L -> End Print R2L

  timing constraints:
    << ... >>
```

Figure 3.9: SP DSL Repeated Motion model - textual representation of the same model depicted graphically in Figure 3.8.

- **Alignables** The relative positions of carriages are defined in terms of Alignables. An Alignable is a physical entity on which the carriage can be aligned, usually a carriage component. Examples of Alignables are “ChannelM” and “ResultImage” in the first entry of the “Begin Print R2L” Situation in Figure 3.9. The same Alignable can be seen in the topmost print carriage alignment in Figure 3.8.

- **Anchor** Defines which part of an Alignable needs to be aligned on. Examples of Anchors are “right” and “left” in the first entry of the “Begin Print R2L” Situation in Figure 3.9.

- **Alignment** An Alignment consists of two instances of an (Alignable, Anchor) tuple, combined with a carriage. It describes the position of a carriage relative to either another carriage or the Result Image. An Alignment can be recognized by the “is aligned with” text in Figure 3.9, which separates the two (Alignable, Anchor) tuples. The carriage contained in the Alignment always supplies one Alignable, while the other Alignable is not from this carriage.

- **Situation** A Situation can be seen as a moment in time where one or more carriages in the printer engine are aligned in a certain way, like a state in a state machine. It consists of one or more Alignments, each of which is annotated with a carriage speed. A Situation may contain one up to n carriages, depending on the number of carriages that are defined in the relevant PSL model. An example of a Situation is “Begin Print R2L” in Figure 3.8 or Figure 3.9.

- **Situations** This is a sequence of Situation instances. The Situations in a Repeated Motion are defined in the sequence in which they are to be executed. The Repeated Motion concept

contains all positional information of a motion strategy. This is a large portion of the behaviour of a motion strategy. Here the visual representation is a great aid in giving insight as to the behaviour of a certain motion strategy, most clearly shown in Figure 3.8.

- **Trajectory** Another concept of importance is a Trajectory. A Trajectory takes a pair of Situations and defines the motion that takes place between these Situations. This can be a constant speed, or a turn. This is another important aspect of a motion strategy.

- **Result Image** An Alignable that refers to the part of the print medium where the image is to be printed. As logic dictates, the edges of where the image needs to start or stop often coincide with the alignment of a printer component. The Result Image can be seen as the absolute reference to which carriages are aligned. This reference only becomes absolute once an actual image is printed. At DSL model level the Result Image can have an arbitrary size.

3.3 Summary

This chapter introduced the embedded software and relevant DSLs as they existed at the start of this project. For both domains, details particular to the carriage motion aspects were given. This gives context to the output and input domains which this thesis aims to connect. Next we look at the options considered to achieve this connection.

Chapter 4

Approach

This chapter shows the approach taken to answer the main research question, that of the feasibility of linking carriage motion DSL models to the embedded software.

4.1 Options for Determining Feasibility

Feasibility is the main focus of this thesis. This raises the question of how to determine whether a problem can be solved or not. A few approaches were considered:

I. Find related work that solves this problem or a similar one.

II. Implement a prototype that integrates the DSL model in embedded software.

III. Theoretically analyze the input and output domain, and either find a mapping based on theory or come up with a counterexample showing that such a mapping cannot exist.

IV. Theoretically analyze the input and output domain, and revise the input DSL language to match the output domain.

The choice was made at an early stage to go for option II. There is a certain risk associated with this strategy; if the implementation that integrates the DSL model in embedded software is successful, feasibility is shown. The reverse however is not necessarily true. If the implementation does not succeed, it could be that the approach chosen for this implementation was not feasible, though another approach that is feasible could exist. Having an outcome of this research stating that a certain approach is infeasible is of lesser value than an outcome that states the entire problem is infeasible.

Option IV was only identified at the end of the project. This option resulted from the execution of Option II, and is akin to Option III with the exception that it aims to provide a solution to possible mismatches between the input and output domain, instead of concluding that the domains cannot be bridged.

4.2 High Level Connectivity Options

In any case, it makes sense to first look at the options for linking the DSL to the embedded software from a high level perspective. The objective is to take behaviour described in a Scanning Productivity DSL model, and to make the embedded software execute this behaviour. The embedded platform runs executable binaries that are compiled from C++ that is generated from RTist models. Some transformation is needed to make the two compatible. We will consider four main alternatives to achieve this. A figure is included with each option. In each of these figures a solid line indicates an existing connection, while a dotted lines is a connection that does not yet exist. Squared edge blocks refer to embedded software entities, while oval shapes indicate a DSL.

4.2.1 Generate an RTist Model

The first option is to connect the DSL to RTist (Figure 4.1). This could be done by generating an RTist model from the DSL. RTist models are EMF models which represent the UML-RT model elements. These are stored in a proprietary type of XML format which is not well documented, and not intended as an API. As such this format may change without notice when RTist is updated. An advantage of this approach is that RTist UML-RT models offer a higher level of abstraction compared to going to C++ directly. This higher level of abstraction may also aid in validation, as analysing a model at a higher abstraction level is easier than analysing generic code. Furthermore, as the generation from RTist models to C++ already exists, this generation step can be reused.

It is also possible to create and modify RTist models using an IDE-based API and use this to connect a DSL model to an RTist model. This could be a viable alternative if Eclipse-based DSL tooling such as XText was used for the DSL. This would involve porting the Scanning Productivity and Print System Layout DSLs to XText. CPP however made the choice to only use MPS for their DSLs so this option was not considered further. In terms of connectivity Figure 4.1 fits with this approach, but the "generate" arrow towards the RTist model annotation might be replaced with "connect".

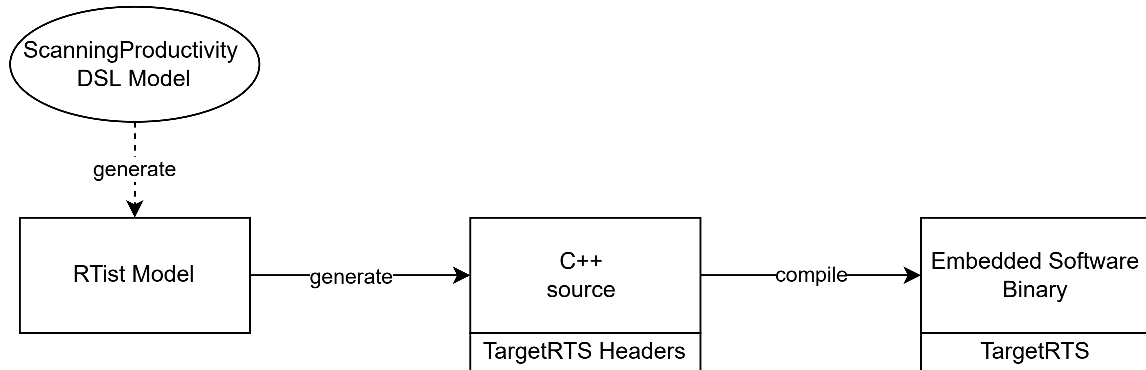

Figure 4.1: High-level overview for option 1, generating a RTist model.

4.2.2 Connect at DSL Level Through ESME

Another option is to use ESME (Figure 4.2) [10]. ESME implements RTist concepts in MPS DSLs. From these DSLs, ESME generates TargetRTS-compatible C++ code. By using ESME, eventually RTist could be replaced entirely. A stepwise approach where RTist capsules not containing motion related functionality are left alone is again a possibility. A major advantage of this approach is a high level of flexibility, as all motion related functionality is available at DSL level. Two way communication between the DSL and ESME would be possible, opening the door to interactions not possible using any of the alternative scenarios discussed here.

ESME is however at proof of concept stage and does not implement all RTist functionality in use in the existing RTist embedded software for the Colorado printers. Another drawback is that ESME does not support generating back to RTist-compatible models, so once an RTist module is imported into ESME, all future maintenance for that model needs to be done inside ESME without the option of falling back to RTist. A modified RTist module could be imported again by ESME but this likely changes interfaces, requiring human intervention in order to fix the changed dependencies caused by this interface change.

4.2.3 Generate TargetRTS Compatible C++ Code

Instead of going through RTist, another option is to directly generate C++ code compatible with the TargetRTS runtime environment (Figure 4.3). RTist capsules that do not contain carriage

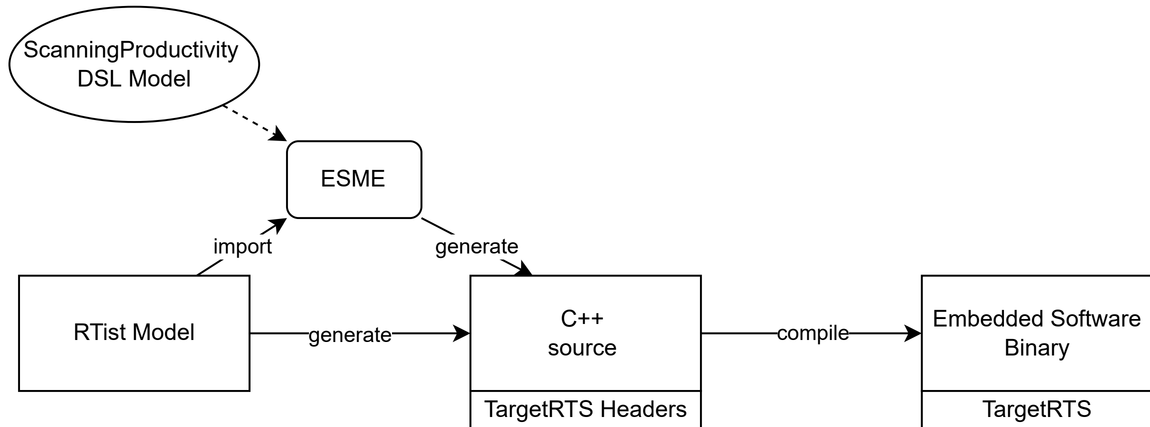

Figure 4.2: High-level overview for option 2, connecting at DSL level through ESME.

motion related code could be left alone, but other modules would be generated directly from the DSL. With this approach however it would be difficult to keep the RTist implementation and DSL generator in sync as there is no well-defined interface between the DSL and RTist models. This implies that whenever an RTist module is modified, the DSL generator needs to be updated manually to reflect these changes.

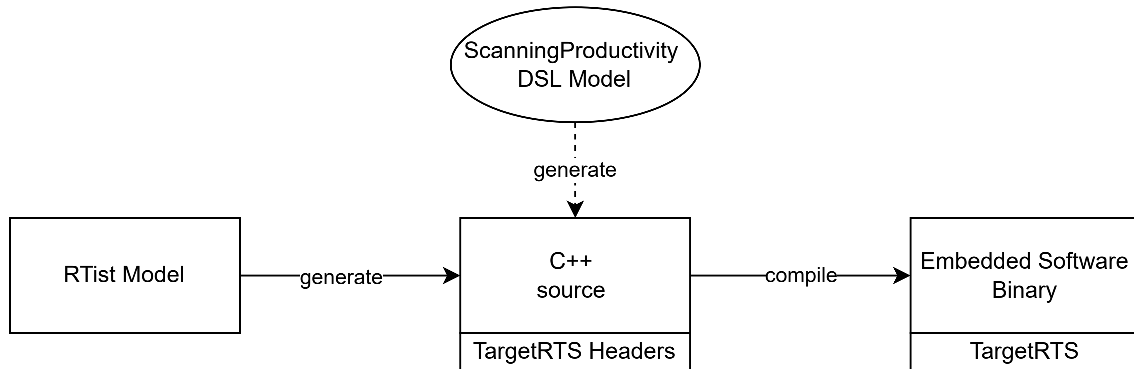

Figure 4.3: High-level overview for option 3, generating TargetRTS-compatible C++.

4.2.4 Generate Stand-alone C++ Library

Rather than directly connecting the DSL to RTist, an option is to generate a separate C++ library from the DSL (Figure 4.4). This library can then be integrated in the relevant parts of the RTist model. A drawback of this approach is that there is no flexibility for modifying RTist models at DSL level; for example altering RTist state machines from the DSL is not possible in this scenario. A major advantage here is a clear two-step approach - generating the library from the DSL can be done separately from integrating the library in RTist. Furthermore, the library could be used in stand-alone fashion which may be useful in design space exploration or for use in other tools that give insight in printer behaviour.

4.2.5 Overview

To conclude we give an overview picture that shows the four main options in context in Figure 4.5. This may help in understanding the differences in the approaches presented.

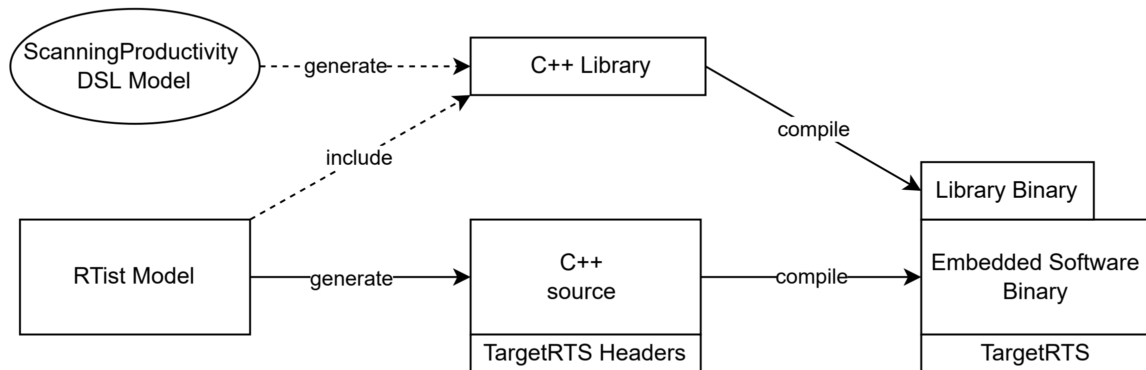

Figure 4.4: High-level overview for option 4, generating a stand-alone C++ library.

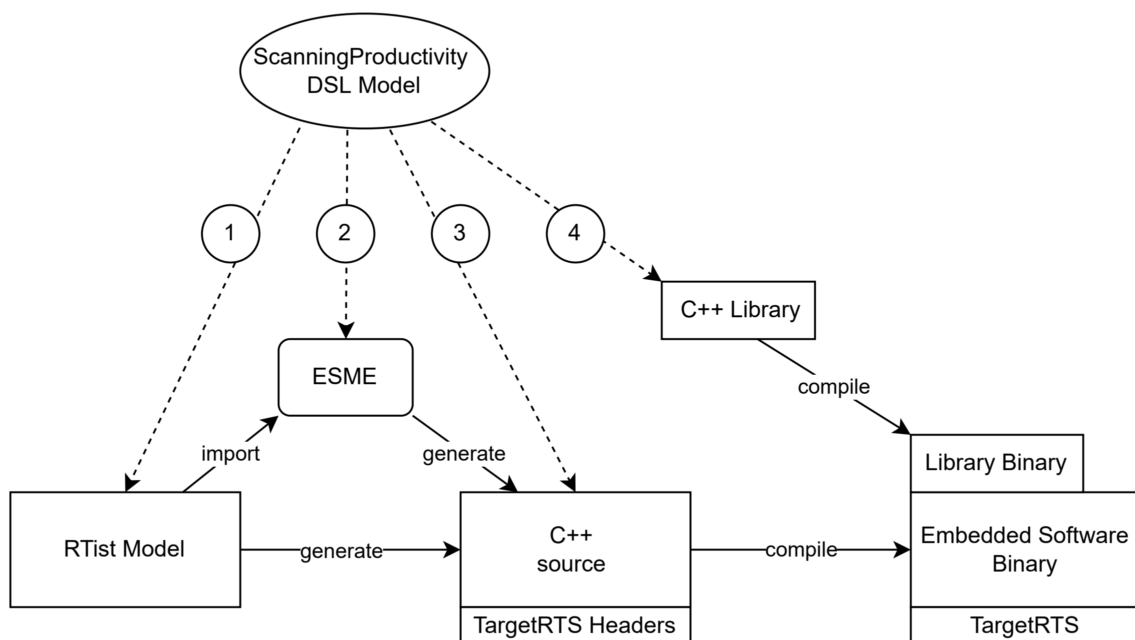

Figure 4.5: High-level overview of the alternatives for connecting the DSL to the embedded software.

4.3 Qualitative Evaluation

To compare the approach options, each is scored in terms of several metrics. The metrics range in desirability from ++ to − −, i.e. from highly desirable to highly undesirable. The results are placed in Table 4.1. The risk and maintainability metrics are considered the most important. The flexibility metric was given a lower importance; if an option has a high risk of not being successful and has poor maintainability, flexibility is hardly relevant. Implementation effort is also of lesser relevance compared to the two most important metrics, though there is an argument to be made for selecting a high risk but low implementation effort option should such an option exist.

	Option 1 RTist model	**Option 2 ESME**	**Option 3 TargetRTS C++**	**Option 4 Stand-alone Library**
Risk	−	−−	−	+
Maintainability	+/−	+/−	−−	+
Flexibility	+	++	−	+/−
Implementation Effort	−	−−	−	+

Table 4.1: Qualitative evaluation of DSL generation target approach options.

4.4 Chosen Approach

From the presented high-level options, option 4, the stand-alone C++ library option, was selected (see number four in Figure 4.5). The idea is to manually write an external C++ library that integrates with the embedded software, which later can be generated from a DSL. This library expresses concepts from DSL models in C++, so generating compatible C++ code from DSL models is possible. The embedded software can then determine the required carriage motion strategy based on information from this library. If it can be shown that the integration between the embedded software and the library is generic enough to support the range of motion strategies that the DSL can specify, feasibility is shown.

In order for this approach to work, the concepts described in the DSL models must map onto the motion execution part of the existing embedded software. Given that it is possible to describe motion profiles in the DSL that are already being executed by the embedded software, at first glance it makes sense such a mapping exists. As we will see in more detail in chapter 5, the DSL can express more than can be mapped onto the embedded software for a specific printer engine. Furthermore, the SP DSL misses the ability to express some concepts used in the embedded software. The initial approach was to ignore this mismatch in expressiveness and focus on Scanning Productivity DSL models that represent motion strategies that exist in the existing embedded software. At a later stage, the SP DSL could be modified to only allow models that can be mapped onto the stand-alone library, and transitively the embedded software.

4.5 Alternative Approach

Modifying the existing SP DSL has its downsides. As we will see in Chapter 5, there is a substantial mismatch between the SP DSL and the embedded software in terms of the way they express motion profiles. Apart from constraints, some additions to the DSL are needed. Most of the required changes are not relevant for the productivity aspect of the language. Being able to describe more concepts than really needed for the specific domain places a higher cognitive load on language users compared to a DSL that is fit for purpose, and defeats part of the reason for using a DSL in the first place.

A more elegant solution could be to design a new DSL that can only specify models that are compatible with the Colorado embedded software. This DSL is denoted as the Colorado Carriage Motion DSL. This would be a DSL that has less expressiveness than the Scanning Productivity DSL, and only allows the specification of models that are compatible with the Colorado embedded software. Colorado Carriage Motion DSL models could then generate to Scanning Productivity models to keep compatibility with tooling that already exists for the Scanning Productivity DSL. The final high-level connections for the chosen approach can be seen in Figure 4.6. Compared to extending the existing DSL, this option has the benefit of allowing concise specification of exactly the behaviour that can also be executed, which in theory would make modeling in this new language easier and more user friendly, while keeping the Scanning Productivity DSL more generic.

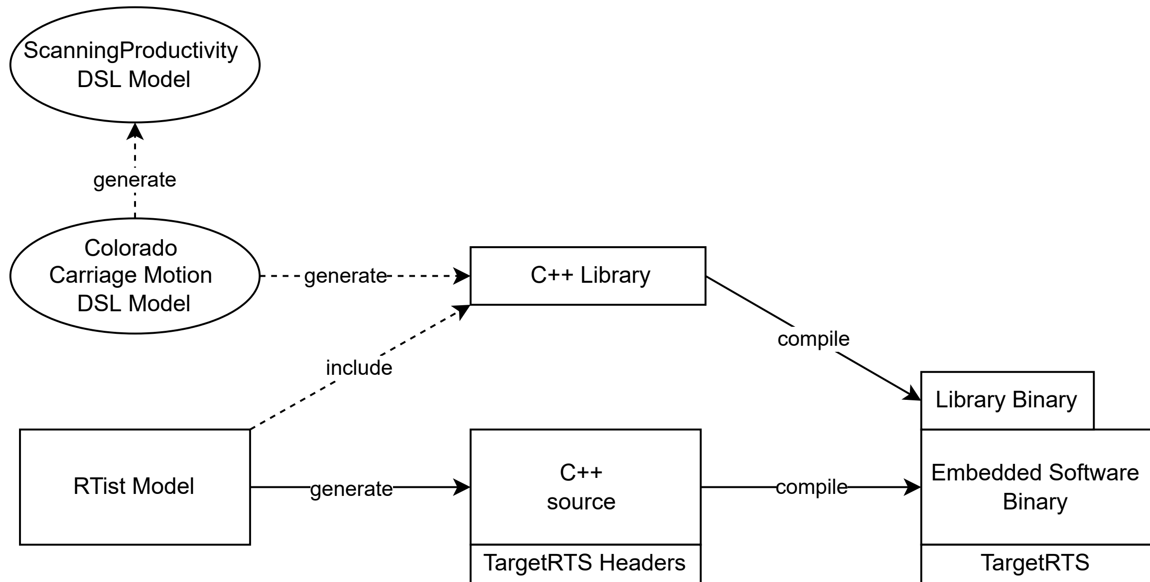

Figure 4.6: Colorado Carriage Motion DSL model and stand-alone C++ Library high-level overview.

4.6 Summary

This chapter considered the possible approaches towards answering the main research question. The chosen approach is to write a standalone C++ library based on DSL concepts that is integrated in embedded software. Lastly a variation of the chosen aproach, where a new DSL is envisioned, was proposed. In the next chapter we will relate the existing Scanning Productivity DSL and the embedded software, to see to what extent their concepts are compatible.

Chapter 5

Relating Carriage Motion DSL and Embedded Software

This chapter analyzes the DSL concepts and compares and contrasts these to RTist concepts to see to what extend these domains can be bridged, and what the semantic gap between them is.

Now that the existing DSL and embedded software are known and the approach is clear we investigate to what extent the concepts in the DSL can be mapped onto the embedded software. The existing DSL was conceived to analyse productivity for scanning-based print systems in general. Our intent however is to specify an executable motion profile for a specific printer. Because of this mismatch in scope, it is possible to define DSL motion profiles that cannot be mapped onto the embedded software. The reverse is true: each Colorado motion profile can be expressed as a DSL model in the existing DSL, though with some caveats.

5.1 Situations Versus Swaths

The main difference between the motion strategies described in the DSL models and the motion strategies as executed by the embedded software is the difference between a sequence of Situations and a sequence of swaths. While the DSL models describe a sequence of relative carriage positions at certain moments, the embedded software executes a sequence of swaths. Swaths have an absolute begin and end position, rather than the relative position defined in a Situation. A swath is also particular to a single carriage, while a Situation refers to one or more carriages.

We could define a Swath in terms of relative positions. Then a Situation could be defined for the beginning of a swath, as well as the ending of a swath. An SP DSL model however contains just a list of Situations. Determining which Situations should correspond to a beginning or ending of a Swath is non trivial. Situations can exist that are neither the beginning nor ending of a swath. An example of a sequence of Situations and a parallel set of swaths is given in Figure 5.1 and 5.2 respectively. Note that the swaths in Figure 5.2 do not end at the same time.

In general, the embedded software schedules a swath for both the print carriage and the cure carriage simultaneously in order to synchronize them. While the embedded software has a mechanism in place to deal with timing offsets, which allows for synchronization of swaths with different start Situations, this is not a concept in the SP DSL. Though the SP DSL does support timing constraints between Situations, still there is no straight forward way of knowing whether these Situations should be scheduled simultaneously or not.

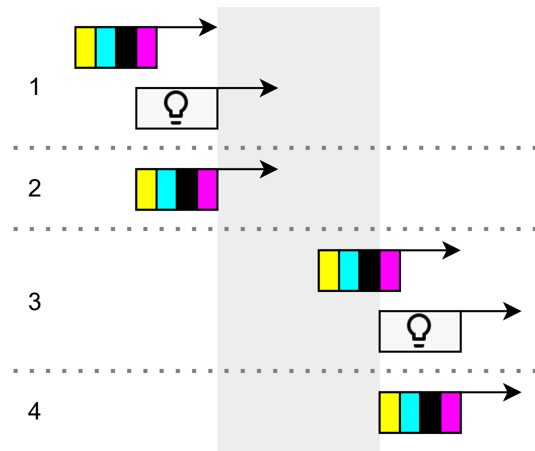

Figure 5.1: An example sequence of four Situations.

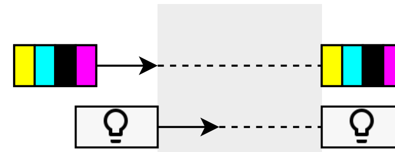

Figure 5.2: A swath of the cure carriage and print carriage corresponding to the four Situations in Figure 5.1.

5.2 Trajectories Versus Swaths

A trajectory describes the movement of a carriage between Situations. A constant speed trajectory is similar to a swath, with the exception of relative versus absolute positioning as described in the previous paragraph. Trajectories in the DSL can denote either a constant speed section or a turn. A turn can not explicitly be scheduled for execution in the embedded software. Any model containing turns therefore can not be mapped onto the existing embedded software. Of course the embedded software in practice does execute turns, but they are a by-product of scheduling swaths. For mono-directional print modes the discrepancy becomes larger, as shown in Figure 5.3 and 5.4. In the DSL, the return trajectory is defined explicitly, while the embedded software is only concerned with planning mono-directional swaths. In the DSL the turns and constant speed return trajectory are specified explicitly, while these are implicit in the embedded software from a swath scheduling point of view. It is worth noting that the SP DSL does not require a user to specify turns in a model. This means it is possible to define a SP model that only uses trajectories that the embedded software could execute.

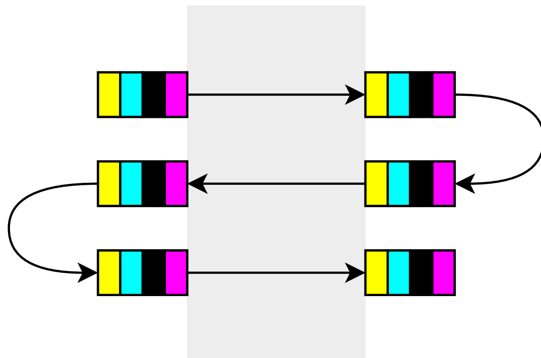

Figure 5.3: An example sequence of the trajectories specified in the DSL to facilitate two mono direction print swaths.

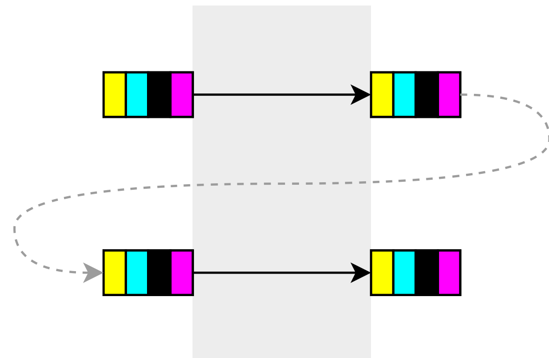

Figure 5.4: The trajectories for two mono directional swaths as required by the embedded software.

5.3 Alignables

In order for the embedded software to be able to execute a motion profile, each Alignable in a motion profile needs to be mapped to a concrete printer component. To be more precise, each Alignable in a motion profile need to be resolved to a specific printer component with a well-defined location and width. This information is needed for a Schedule to be executable by embedded software. In the existing embedded software, the begin and end position for a swath are also calculated in terms of locations of printer components on a carriage. This, however, is much less explicit than in a Situation in the SP DSL.

5.4 Parametrized Alignables

The embedded software has some parameters that can be applied to a selected motion strategy. For example, it is possible to enable the scanner, which is placed at the edge of a carriage. The scanner may need to pass over the entire printed surface. This means the carriage possibly needs to execute a wider movement in order to facilitate the scanning action. Otherwise the motion strategy does not change. The Scanning Productivity DSL currently does not support parametrizing Alignables. So, to support the same motion strategy with the scanner enabled as well as disabled, two separate DSL models are needed. The embedded software also needs to take care that the correct version of the translated DSL model is selected depending on the print mode.

5.5 Synchronizing End Situations

The SP DSL allows synchronization of carriages at the end of a swath. The embedded software however can only synchronize carriages at the beginning of a swath. An example of an SP model that cannot be executed by the embedded software is shown in Figure 5.5.

Figure 5.5: Example of an SP model that cannot be executed by the embedded software.

Of course, if the synchronization at the end of a swath is identical to the synchronization at the beginning of a swath, a side effect is that these carriages will still be synchronized at the end of a swath if they were synchronized at the beginning of a swath.

5.6 Component Order

Related to parametrized Alignables, the SP DSL allows the specification of multiple Situations along a trajectory. At DSL model level however these Alignables do not have a defined size, nor is the ordering of Alignables known. As discussed in Section 5.3, this information is only available at runtime. This causes an issue when multiple Situations are defined for the beginning or ending of a swath. In the DSL model, one of these Situations will be above the other, indicating that the top Situation will occur before the bottom Situation. This is not always the case; it is possible that at runtime the layout of the printer causes the bottom Situation to occur before the top one.

Figure 5.6: Example of two Situations that may not always be in chronological order.

For example, in Figure 5.6 two Situations are depicted, with the “Begin Cure R2L” above “Begin Pincure R2L”. Given the ordering of components on the carriages as depicted, the ordering of the Situations is in chronological order. However, if the Pin Cure Light in the bottom Situation (Marked with dotted line) were to be moved from the right side to the left side of the carriage, the “Begin Cure R2L” Situation should occur before the “Begin Pincure R2L” Situation. The semantics of the order of the Situations, where this order conforms to the chronological order in which these Situations occur, do not always hold. Only at runtime, when component sizes and locations are known, can it be determined which of these Situations should occur first, which changes the component that the beginning of a swath should be aligned on.

5.7 Summary

This section considered the gaps between motion profiles expressed in the SP DSL compared to motion profiles in embedded software. The main conceptual gap is the lack of a swath concept at DSL level. The presented gaps provide an overview of some of the challenges that need to be overcome to bridge the DSL and embedded software domains. Keeping the gaps found here in mind, we can now construct a prototype library that aims to bring the input and output domain closer together.

Chapter 6

Prototype

In this chapter, the implemented C++ prototype library is given in detail. This library defines concepts that encode carriage motion profiles. The main function of the library is to define a motion strategy in terms of a Schedule, which consists of a repeating sequence of swaths. Carriage positions are not related directly, but are related in terms of components that are placed on the carriage. These components are defined in a layout. At runtime the library calculates absolute carriage positions by combining the actual component sizes defined in the embedded software together with the location where the image is to be printed.

This library was written with the intent to generate it from DSL models at a later stage. For this to be possible, the differences between the SP DSL and the embedded software as discussed in Chapter 5 need to be overcome. In particular, three assumptions were made. First, that it is possible to link the correct Situations to a Begin of Swath Situation and End of Swath Situation. Second, that SP models that are converted to C++ library code do not contain behaviour that is not supported by the library. Third, that the order of carriage components is known at model level. The first could be satisfied by extending the SP DSL with a Swath concept, where this linking is done manually by a DSL user. The second could be enforced by constraining said DSL. The third is something a modeler has to take into account when modeling a motion strategy.

6.1 Mapping DSL Concepts to Prototype

As explained in Chapter 5, the existing Scanning Productivity DSL does not map onto the existing embedded software in its entirety. Still, the basic idea of defining a motion profile as a sequence of Situations is upheld. Some concepts in the SP DSL have a closely related relative as a class or module in the prototype library. A UML class diagram of these library concepts is given in Figure 6.1.

The main library classes closely resemble concepts from the SP DSL. The Schedule combines the Repeated Motion, Trajectory and Media Step concepts; each entry in the library Schedule contains either a Media Step, or a combination of 1 or more constant speed Trajectories. Such an entry is called a ScheduleEntry in the library. A Schedule can have any finite number of ScheduleEntries. Trajectories in turn are linked to Situations. The combination of these constant speed trajectories together with Situations is referred to as a SituationSwath in the library. With n being the number of carriages in the system, each ScheduleEntry can have at most n SituationSwaths. Each SituationSwath is associated with exactly one Carriage. Each SituationSwath refers to two Situations - one for the beginning of a swath, and one for the end of a swath.

Situations in turn consist of Alignments. An Alignment refers to two Alignables combined with an Anchor point for that Alignable. A Situation can contain at most n Alignments.

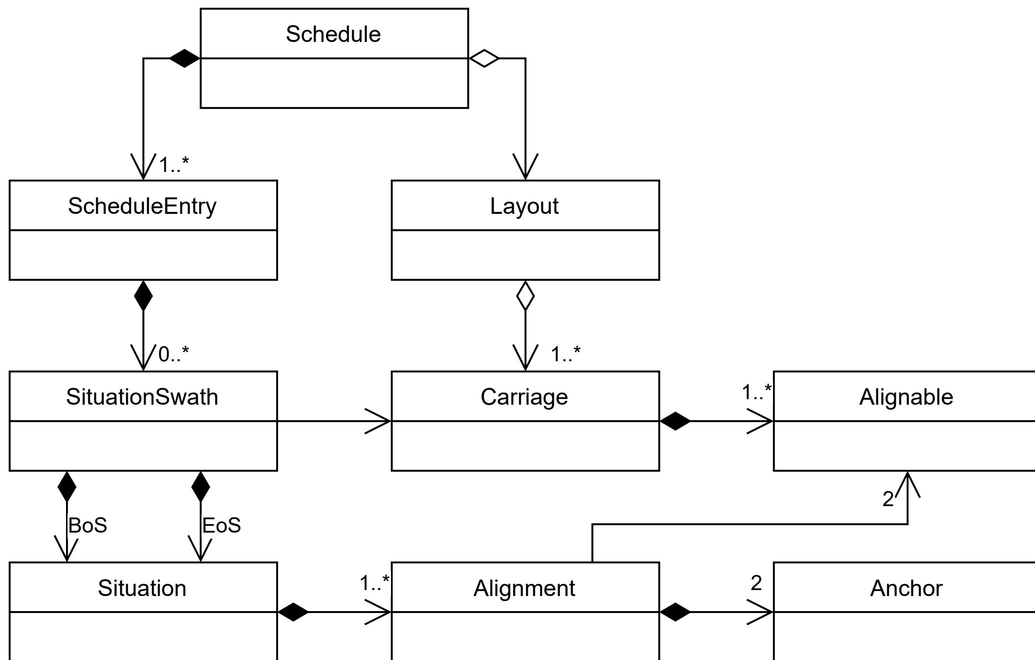

Figure 6.1: Class diagram of prototype library.

6.2 Integrating Schedules in Embedded Software

The prototype library was integrated in the printing part of the embedded software. For reference, Section 3.1.3 describes the flow of planning and executing swaths in the existing embedded software. This process was modified to take the positional begin and end positions for swaths for each carriage from the library Schedule, instead of calculating them inside the embedded software. The remaining part of the embedded software was mostly untouched.

A textual description of the changes compared to the existing embedded software is given, as well as a sequence diagram where the changes after integrating the library are highlighted. This revised sequence diagram is shown in Figure 6.2, with changes shown in red. Alternatively, to view the changes on a black and white printout of this thesis, one may refer to the original sequence diagram in Figure 3.4.

In more detail, in the original embedded software the Data Path sends a print swath request to the Print Procedure. This print swath request contains the positions where the print swath needs to start and end. These positions are not used anymore - instead the Result Image positions, which the Data Path also passes to the Print Procedure, are fed to the library. The library in turn calculates the begin and end positions for the requested swath based on the selected Schedule. A similar modification is done to the swath positions for the cure carriage; instead of calculating the cure carriage swath positions in the embedded software based on the selected cure mode, the swath positions are calculated in the library and fed to the Cure Swath Procedure directly.

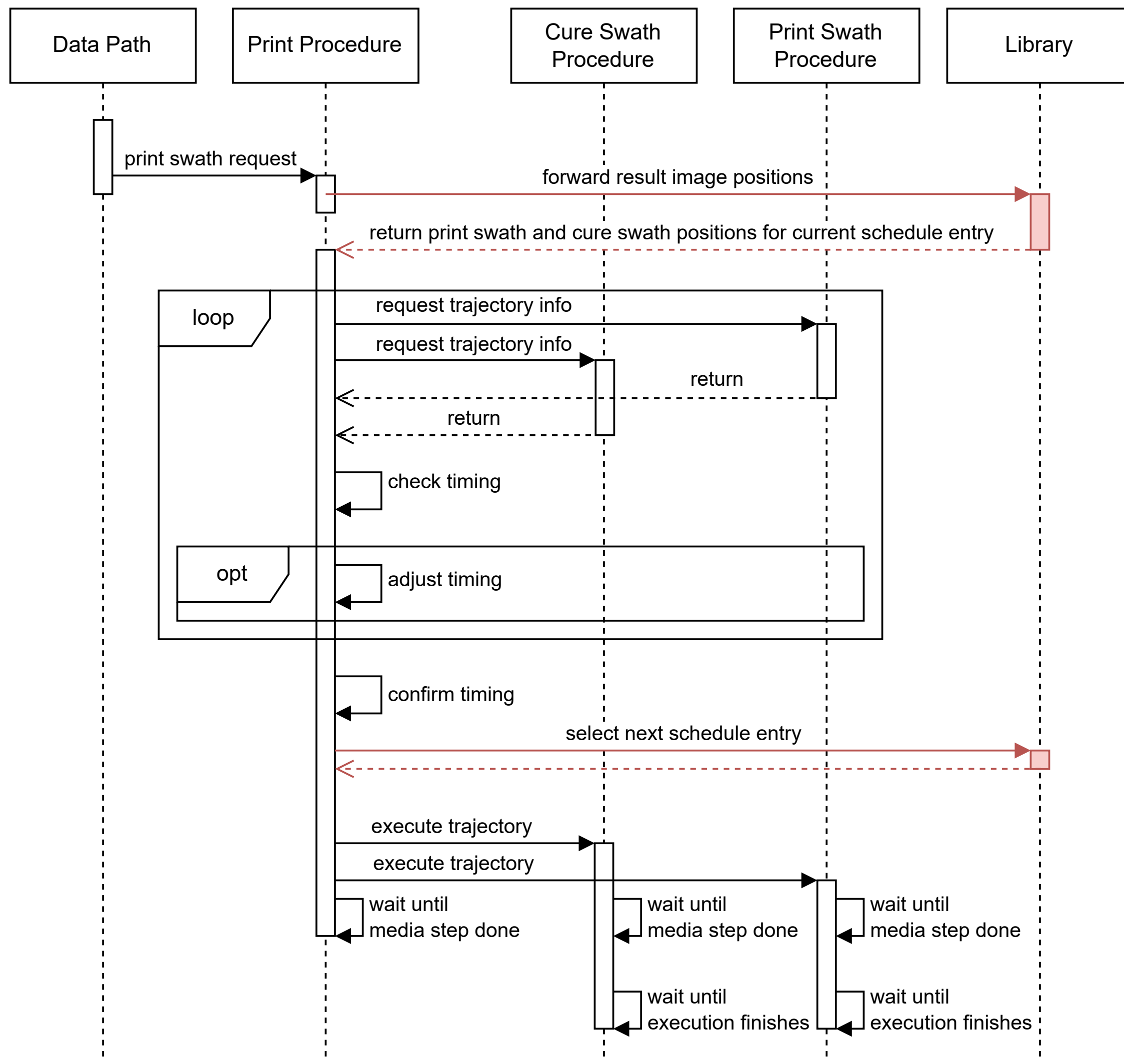

Figure 6.2: Sequence diagram depicting a single iteration of the steady state printing loop*.

6.3 Interfaces

To integrate the library in the embedded software, the library needs to interface with said software. Three main tasks need to be completed:

- The Alignables in the print system layout used in the Schedules need to be linked to actual printer component locations and widths.
- The correct Schedule needs to be selected based on the print mode.
- The print head distance margin needs to be passed.

These tasks require manual actions to satisfy the requirements for these interfaces in the embedded software. In the case of the layout, each Alignable in the layout required by the Schedules needs to be instantiated with a width and a location. This information can be queried

*Please note this diagram is a simplification of reality, and for instance does not take the Action Control Manager (ACM) into account. The ACM handles the plumbing between the different actors and allocates free Swath Procedure instances to the Print Procedure.

from the printer engine database, an assigned to the correct Alignables. Once the layout is instantiated, this layout can be passed to the Schedule instance corresponding to the current print mode. Selecting the Schedule to be used can be done by evaluating the print mode that is selected. In the current implementation this does requires manual implementation in the embedded software for each new Schedule that is added. The final part of the interface is a parameter that the embedded software needs to pass to the library regarding the print head margin. This margin needs to be added between the print heads and the Result Image when the print heads approach the Result Image, and Alignment between these two Alignables takes place.

6.4 Summary

This chapter introduced the prototype library, which encodes motion profiles in a Schedule. It considered the connections between library concepts and DSL concepts, the integration of the library in embedded software and the interfaces needed by the library. In the next chapter we will experimentally verify the correctness of this prototype library.

Chapter 7

Evaluation

In this chapter the prototype library that was integrated in the existing embedded software is evaluated. First the options for verification are laid out, followed by the verification plan. Subsequently we show experimental results together with a discussion of those results. Finally we go over some limitations of the prototype library.

7.1 Verification Plan

Verification of the prototype is done using the Software-In-the-Loop (SIL) simulator introduced in Section 3.1.4. The main mode of verification is to run a SIL simulation for a print with a certain motion strategy on both unmodified embedded software, and a derivative of this software that integrates the prototype library. The carriage motion behaviour between these two is then analysed both qualitatively and quantitatively using Jupyter Notebooks [6] with the aid of MatPlotLib [4], NumPy [2] and SciPy [14]. The swath positional data was gathered from the logging output of the embedded software.

In theory, the behaviour of the existing and modified software should be (near) identical. If this is the case, it suffices to compare the calculated BoS and EoS position for the different swaths, together with the productivity metric. If these values agree, then the modified software using the library is functionally correct, and no further analysis is needed. In the end, all the library does is calculate BoS and EoS position for swaths. If BoS and EoS positions differ from the original instructions, or productivity has changed, further analysis is required.

Two motion strategies are considered; (1) a normal curing motion strategy where the cure carriage moves in counter-phase with the print carriage, and (2) a pin curing mode which places tighter requirements on carriage synchronization. These motion strategies were manually encoded as two Schedule classes in the prototype library. SP representations of the encoded motion profiles can be found in Appendix A.

The chosen print job was for an image of 1 meter wide by 16 centimetres high. This width is wide enough to refrain from triggering the minimum swath width limitation*, and the length is long enough to have a sufficient number of swaths to perform our analysis on.

7.2 Verification Options

This section enumerates the options that were considered to verify the correctness of the experimentally implemented prototype library.

*When printing narrow images, e.g. of a candle, the carriages execute wider swaths than necessary from a functional standpoint, which is done to prevent overheating of the drive motors.

7.2.1 Unit Tests

As the prototype library can be compiled to a stand-alone executable, it is possible to run unit tests on (parts of) the software. This was used during the development of the library to ensure correctness of the different modules inside the library. This includes some alignment calculations that output carriage position, which are compared to carriage positions as output by the embedded software executed by the SIL simulator. While unit tests execute quickly, it is difficult to make sure the correct embedded software data is linked to a particular alignment calculation this way.

7.2.2 Simulator

Taking the ideas of the previous paragraph a step further, an option could be to write a simulator, not to be confused with the SIL simulator, that executes a Schedule from the prototype without involving the embedded software. Possibly this simulator could use the same carriage trajectory calculation as used in the embedded software. This is another lightweight option that likely calculates the start times and positions of a print job in seconds, or at most a few minutes. Only a single ground truth SIL run would be needed per print mode. However, in case the simulator result does not agree with the ground truth, analysis could be difficult, as mismatches could be due to library errors or simulator errors.

7.2.3 Embedded Software Logging Data

The embedded software produces output for several types of carriage motion related data. We can use SIL to run different versions of the embedded software, which allows a detailed comparison between the ground truth, the unmodified embedded software and the modified prototype library software. The main drawback of this option is the execution time; it takes about 45 minutes to execute a simulation for a single print job on an 8th generation Intel i7 Windows laptop. Other options provide a much faster turnaround time but do not provide the same level of confidence for comparing against the ground truth.

Some produced data that is relevant to carriage motion:

Per-Swath Data

The embedded software generates a file which contains detailed information of timing and begin- and end positions of each swath. This gives quantitative data to compare the calculated swath positions for modified versus original software.

High Resolution Positional Data

Apart from swath information and productivity information, a more detailed positional carriage data is generated. The setpoint of each carriage control loop is logged each iteration of the control loop, which executes at 1kHz. This can be used to give both quantitative and qualitative insights in the motion executed by the carriages. This data is only relevant if the trajectories calculated between original and modified software differ.

Productivity

Another example of data that is produced is a productivity number, which lists the square meterage per hour of images that were printed (virtually). This way it can easily be checked to what extent productivity is influenced by integrating the prototype library. Being just a single scalar number, it does not give insight as to why productivity is changed. Furthermore, identical productivity number does not guarantee identical behaviour. Given that productivity is a key aspect in carriage motion, it still makes sense to consider this number.

Unstructured Logging Output

The embedded software writes generic logging output to a text file. This includes the positional and timing information for swaths. The log lines containing this information can be traced to the embedded software source code producing those log lines. This source of data therefore is the most reliable in the sense that we can be sure we are using the right data. A drawback of using logging data is that the relevant information is buried in between tens of thousands of lines of irrelevant log entries. Some relevant information however is not available using other embedded software data sources.

7.2.4 Physical Printer Engine

Verification could be done on a physical printer engine. This would generate the same log files that the SIL simulator generates, so any analysis done here could be replicated with real engine data. Furthermore, one could observe the physical carriage movements. However we are only interested in the calculated swath positions, and the setpoints to carriage drive control loops, which do not change when going from a SIL simulation to a physical engine; we will argue why this is the case in the next paragraph. As the data does not change, it does not make sense to use an expensive physical machine when an alternative like the SIL simulator is available. Another reason for not using a physical engine is that in case an error occurs, it is possible that uncured ink may leave the engine. As uncured ink causes a potential safety hazard, this situation is best avoided.

7.3 SIL Simulator Suitability Justification

As stated in Section 7.1, in cases where analysing calculated swath positions is not sufficient, we look at carriage movement simulated by the SIL simulator. While this simulated carriage movement is a simplification of real carriage movement, this is not relevant for our analysis. We will give an argument for this based on two assumptions.

First, the carriage control is position-based. In other words, the carriage position setpoint is always used as a basis for control, and there is no feedback to this setpoint. Therefore, when a motion trajectory is calculated by Calculate Trajectory, the end position as calculated ahead of time can be used to calculate the next trajectory without having to execute any trajectory. Here we do assume that the carriage control loop is able to execute the trajectory that was calculated. If correct values for maximum speed and acceleration are input when calculating the trajectory, this assumption holds in practice. In case this assumption does not hold this implies a problem with the carriage control process, which is out of scope for this thesis.

Second, the prototype library does not execute complicated long running calculations. The way the SIL simulator works, unlimited CPU time is available to the application. It is possible for long running calculations to cause timing issues on a real printer engine. The calculations performed by the prototype, however, are lightweight in nature, so no such issues are expected. Moreover, no significant change in execution time of SIL simulations between the original and modified software was observed.

7.4 Experimental Results

Here the results of the evaluation is given for the two print modes considered. We will first take a look at the results for the default print mode, followed by those for a pin cure print mode[†].

7.4.1 Default Print Mode

In this print mode the cure carriage and print carriage move in counter-phase. The Schedule repeats after two ScheduleEntries containing swaths. So, after two ScheduleEntries are executed,

[†] A Leading Pin pin cure print mode to be more precise.

the Begin of Swath (BoS)/End of Swath (EoS) pattern repeats until the entire image is jetted.

All positions that are calculated using the library are identical to the positions calculated in the original software. To be more specific, the BoS and EoS position for the cure carriage, as well as the BoS and EoS positions for the print carriage, are identical for all swaths in the repeating pattern. As expected, the productivity does not change either.

7.4.2 Pin Cure Print Mode

In this pin cure print mode, the cure carriage and print carriage move in phase. The pin cure mirror on the print carriage and the pin cure light on the cure carriage need to be synchronized. The Schedule again repeats after two ScheduleEntries containing swaths. So, after two entries are executed, the BoS/EoS pattern repeats until the entire image is jetted. The swath begin- and end positions for both these entries are given in Table 7.1 and 7.2. The productivity of 15.053 m2/h is unchanged between the original and modified software.

Taking a look at the print carriage BoS and EoS positions, we see the library positions are identical to the original software. In case of the cure carriage, the BoS position for even swaths and the EoS position for odd swaths differs. The absolute difference between the library and original positions in both cases is the same at 0.2215 m. Looking more closely, we observe the original even BoS position is identical to the library EoS position, and the original odd EoS position is identical to the library BoS position. This is shown visually in Figure 7.1 and 7.2. This difference of 0.2215 m is exactly the distance between the left edge of the left pin cure mirror, and the left edge of the pre cure light. It seems that the original software wants a swath to start at a certain printer component, while the library wants the previous swath to end at this component.

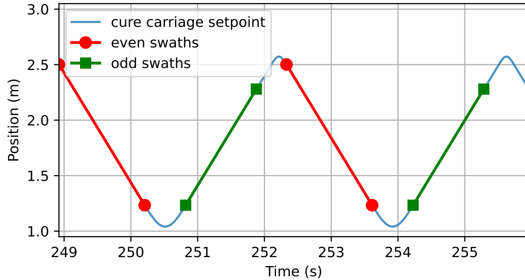

Figure 7.1: Pin cure swaths for the first four swaths of the print job - original software.

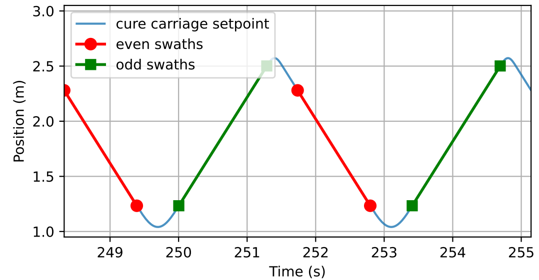

Figure 7.2: Pin cure swaths for the first four swaths of the print job - modified software.

	Original	Modified	Difference
BoS Print	2.64782	2.64782	0
EoS Print	1.403285	1.403285	0
BoS Cure	2.501320	2.27982	0.2215
EoS Cure	1.233474	1.233474	0

Table 7.1: Pin cure print position setpoints (m) - even swaths.

	Original	Modified	Difference
BoS Print	1.219974	1.219974	0
EoS Print	2.463509	2.463509	0
BoS Cure	1.233474	1.233474	0
EoS Cure	2.279820	2.50132	-0.2215

Table 7.2: Pin cure print position setpoints (m) - odd swaths.

To trace the source of this discrepancy between the original and modified software, we take a look at more detailed positional SIL data. We use the positional setpoints to the high-frequency carriage control loop for this analysis.

Carriage Alignment

First we need to check if the pin cure carriage alignment is executed correctly. We plot the first four swaths for the print job for both software versions in Figure 7.3 and 7.4. The data is plotted in terms of the pin cure mirrors and the pin cure light to see if this alignment is correct.

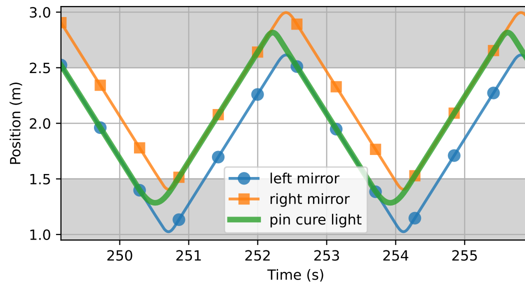

Figure 7.3: Pin cure position setpoints for the first four swaths of the print job - original software.

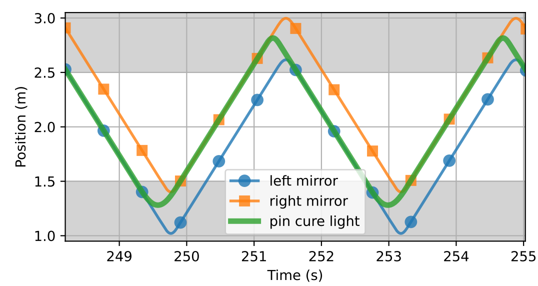

Figure 7.4: Pin cure position setpoints for the first four swaths of the print job - modified software.

In both cases, visually the alignment appears correct. The alignment pattern is the same - when going from high to low positions (or right to left), the pin cure light is aligned with the left pin cure mirror, while alignment takes place on the right mirror when travelling in the opposite direction. The white part of the graph represents the location of the Result Image, the part of the print surface where ink is jetted.

To quantify how well the carriages are aligned, we take the control loop position setpoints of the pin cure light and the relevant mirror when these are moving over the print surface while jetting, for a print job consisting of 36 swaths. Then we take the absolute difference between these setpoints and calculate the maximum, mean and median misalignment, the results of which are listed in Table 7.3.

	Original	Modified	Difference
Maximum	0.0480	0.0460	-4.2%
Mean	0.0167	0.0160	-4.2%
Median	0.0150	0.0140	-6.7%

Table 7.3: Carriage setpoint misalignment (mm) while jetting - entire job consisting of 36 swaths

We see the library misalignment is on average 4% lower than the original software. Some misalignment between the light and the mirrors is to be expected, as these setpoints are expressed in terms of position encoder values, which have a limited resolution. In any case, the library pin cure alignment during execution is at least as good as in the original software.

It seems that even though some calculated BoS and EoS positions differ, the functional behaviour of the carriages is correct in the library, and the productivity is unchanged compared to the original software. This may seem odd, but can be explained by two aspects of carriage movement:

1. Turns are symmetrical if the magnitude of the speed before and after the turn is unchanged.
2. Between the position a turn ends and the BoS position, as well as between the EoS position and the begin position of a turn, the carriage speed is equal to the swath speed.

The first is the case because swaths in both directions have the same constant absolute speed. The second follows from the way trajectories are calculated, as explained in Section 3.1.3. This means that, from a carriage motion perspective, it does not matter if a swath ends at a certain alignment, or the next swath begins at this alignment. To illustrate this, Figure 7.5 and 7.6 highlight such a turn. Effectively these figures show a close-up of Figure 7.3 and 7.4.

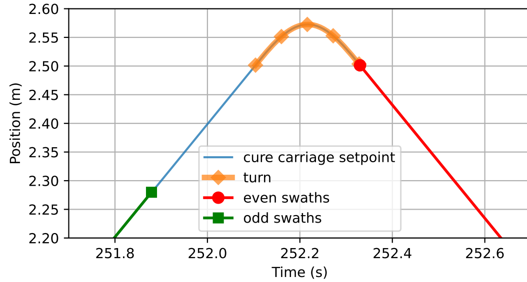

Figure 7.5: Pin cure position setpoints for the first odd to even swath turn - original software.

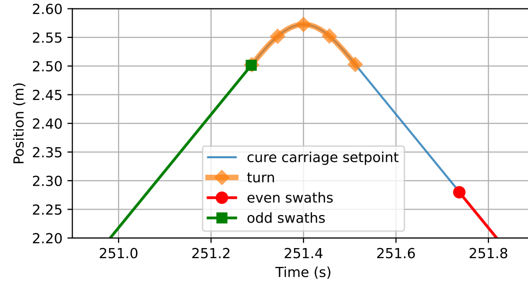

Figure 7.6: Pin cure position setpoints for the first odd to even swath turn - modified software.

7.4.3 Replicating Original Pin Cure Alignments

In the previous section we hypothesized that the original software is aligning on a different printer component in the cases where the calculated positions do not agree. To check this hypothesis, we modified the pin cure Schedule in the library to start odd swaths on the precure light, and to end even swaths on the edge of the pin cure mirror. SP representations of this Schedule are again available in Appendix A. And indeed, using this modified pin cure Schedule, the BoS and EoS positions calculated by the library are identical to those in the original software.

However, for even swaths, the pin cure alignment is now off. This is because the library does not support delaying swaths. This limitation will be discussed in more detail in Section 7.5.1. Furthermore, the simulated engine only executes two swaths before aborting the print job. This is due to a combination of the aforementioned delay issue, and an issue with the timing of cure lights, which is another limitation of the library which will be discussed in Section 7.5.2. The embedded software detects the misalignment between the pin cure light and the pin cure mirror and aborts the print job. The first two swaths for the modified software are given in Figure 7.8. The first two swaths of the original software in Figure 7.7 serve as a reference.

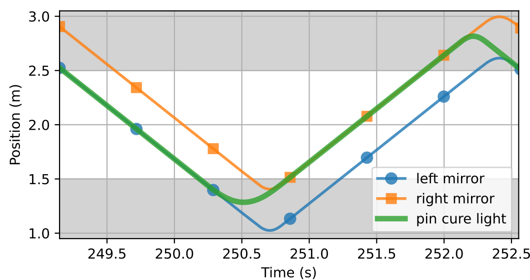

Figure 7.7: Pin cure position setpoints for the first two swaths of the print job - original software.

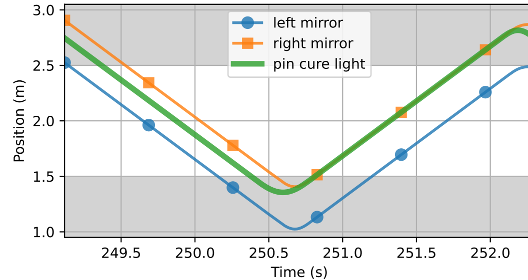

Figure 7.8: Pin cure position setpoints for the first two swaths of the print job - modified software - modified pin cure model.

We indeed observe a misalignment for even swaths. The odd swaths visually appear to still be aligned. This is expected, since the BoS positions for the odd cure swaths were not changed

compared to our original pin cure model, for which alignment was already shown in Section 7.4.2.

In case of the even swaths we expect the misalignment between the pin cure light and pin cure mirror to be near the 0.2215 m position difference we observed in our original pin cure Schedule. Like before, we calculate the distance between these components for the single even swath for which we have data. We only consider the samples where the pin cure mirror is above the Result Image, in total about one thousand samples. For each sample of the pin cure mirror location, the sample that is closest in time for the pin cure mirror is found, after which the distance between these components is calculated. Indeed we find an average misalignment of 0.22152 m, with a median of 0.22153 m. All values are in the range of [0.22149,0.22154] m. The error between the expected and observed alignment is in the 0.02-0.03 mm range, which is in line with the errors observed on correct alignment such as in Table 7.3.

7.5 Limitations

This section discusses some limitations of the implemented prototype library.

7.5.1 Delayed Start

One limitation of the library is that two swaths that need to be synchronized need to start at the same moment in time. The original software does support delayed start when synchronizing swaths. This means some productivity can be lost when using the carriage motion library in its current state. It is possible a carriage needs to be aligned with another carriage at an earlier moment than strictly necessary from an alignment point of view. Figure 7.9 and 7.10 show the difference in BoS timing between the original and modified software. The vertical dotted lines highlight the timing difference between the two.

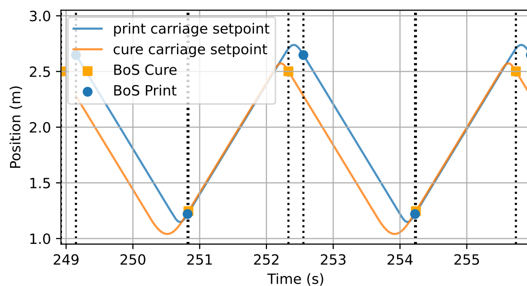

Figure 7.9: Pin cure BoS timing - original software.

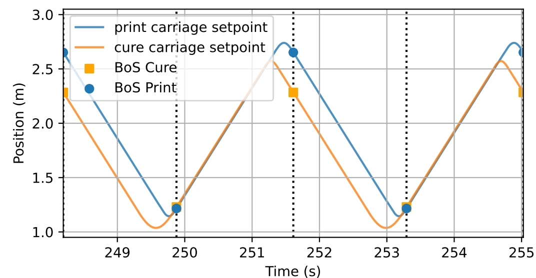

Figure 7.10: Pin cure BoS timing - modified software.

7.5.2 Light Timing

As discussed, the prototype library replaces the existing carriage movement functionality in the embedded software. The timing of turning on or off the different cure lights on the cure carriage is a part of this existing functionality. This timing was not implemented in the prototype library. Although the lamp timing was coupled to the BoS and EoS positions in most cases, for pin curing adjusted moments were used. These adjusted moments are not calculated (yet) by the prototype library. Therefore, the timing of the lights on the carriage is not working correctly in the modified software.

7.5.3 Evaluation Scope

Two motion strategies were considered in the verification in this chapter. While the chosen strategies are representative for the type of print modes that are commonly used, motion strategies

that incorporate slow curing or mono-directional printing were not evaluated. In both cases a manual library implementation of such a motion strategy was attempted, but the execution of these was unsuccessful. A contributing factor to this failure is the incorrect lamp timing as mentioned in the previous paragraph. The lamp timing error is detected by the embedded software which subsequently cancels the print job.

The evaluation is also limited to a single configuration of a Colorado printer from the Colorado printer family, which consists of several engines that vary slightly in their layout. Even though the principles of the library conceptually are the same for these different engine configurations, the evaluation could have included more engine configurations.

7.5.4 Manual Implementation

All Schedules in the prototype library were implemented manually. Although Chapter 6 explains the library is based on concepts from the Scanning Productivity DSL as introduced in Section 3.2, and the intent was for this library to be generated from DSL models, no automated transformation from DSL models to prototype library Schedules was implemented.

7.6 Summary

After considering the verification plan options and discussing the considered verification options, experimental results of the prototype library were evaluated. We observed that the prototype library is capable of synchronizing carriages and executing the same resulting motion behaviour as the existing embedded software for the two evaluated motion strategies. Finally several limitations of the library were considered, one of which is the manual implementation of library Schedule instances. In the next chapter we will see some recommendations for the design of a new language for describing carriage motion profiles, which could be a source language for automating the generation of these prototype library Schedules.

Chapter 8

Colorado Carriage Motion DSL Outline

This chapter sketches the outlines for the Colorado Carriage Motion (CCM) language, based on the insight gained during this research.

Though this thesis mostly focussed on the Scanning Productivity language as a basis for modeling carriage motion strategies for the Colorado printer, this language is not well suited to the task of deriving an executable version of these models for use in the embedded software. As was alluded to in previous chapters, a better approach is to design a new language that more closely matches the realms of the embedded software. This language could reuse concepts from the existing DSLs discussed in Section 3.2, so that a Schedule from the prototype library can be generated from models in this new CCM language.

While some initial steps towards implementing this language in MPS were undertaken, due to time constraints no finished language can be presented here. This research however did uncover a number of things that should be taken into account when designing this language. This is mostly based on the incompatibilities between the Scanning Productivity DSL and the Colorado embedded software as discussed in Chapter 5, and partly on general realizations that manifested when working with the Scanning Productivity language for a longer period of time during the course of my internship at CPP.

What follows is an itemized list of several matters that should be taken into account when engineering the Colorado Carriage Motion language.

- Swaths should be modeled explicitly. This includes a direct description of synchronization of swaths between carriages, and the ability to delay one swath relative to another swath.
- Swaths should be reusable. Print swaths for different print modes often use the same swath definition. This currently causes duplication of models that could be prevented.
- It makes sense to add annotation of functionality that is closely related to carriage motion. The timing of enabling and disabling the cure lights is one example of such functionality.
- The components on which swaths can be aligned should be configurable. This prevents duplication of motion strategy models with only minor differences. This configurable component would take a parameter that decides which component is used during execution, based on the software configuration for the print job that is executed.
- A mechanism for dealing with the order in which components are placed on carriages is needed. For instance, an “outermost” concept might be added that refers to several components. At execution time, when the printer layout is known, this concept could be resolved to whatever component from “outermost” is placed furthest on the carriage.

- An abstraction layer between a specific Print System Layout and the components used in a motion strategy should be considered. Many motion strategy models use only a subset of components from a Layout. An abstraction to this Layout would allow reuse of motion strategy models between different Layouts that share this common subset of components. This recommendation was taken over and implemented in the SP language during the course of the internship.

Chapter 9

Conclusions

To summarize, a prototype library was implemented which manually encodes carriage motion profiles based on DSL carriage motion concepts in a C++ library. This library in turn is accessed by the embedded software to execute these motion profiles. Finally the execution of these library-based motion profiles was analysed.

We will now answer the research questions by first revisiting each question, followed by a reasoning leading to an answer. We start with the main research question.

> **RQ1**: Is it possible to use Model Driven Engineering to transform carriage motion DSL models to RTist-based embedded software?

Even though Chapter 7 shows the implemented prototype has its limitations, it was shown that it is possible to capture DSL-based carriage motion concepts in a library which the embedded software can use to execute a carriage motion profile. Chapter 5 however showed significant conceptual differences between the SP language and the embedded software. With particular care it was possible to manually construct SP models that closely resemble motion profiles in the prototype library, and in turn the embedded software. Nevertheless, automating this construction from generic SP models to prototype library artifacts is not possible using the SP language in its current state.

As the SP language was found to be unsuitable for specifying carriage motion DSL models compatible with the embedded software, we considered an alternative. In Chapter 8 we sketched the outlines for a new DSL that is at the same conceptual level as the embedded software. We are confident that, using the insight gained in this research, we can construct the Colorado Carriage Motion language to overcome the conceptual differences between the SP language and the embedded software that were encountered in this research. This confidence together with the results of the implemented prototype library leads us to positively answer the main research question in this thesis.

> **RQ2**: What are the main challenges in translating models expressed in CPP's Carriage Motion DSL developed in JetBrains MPS to RTist models?

Looking back, the most difficult part in aligning the DSL and embedded software domains was to get a clear definition of concepts used in the embedded software, and to determine the compatibility between these embedded software concepts and existing DSL concepts. While at a glance some concepts seemed to be compatible, when diving into the details it was repeatedly found that a concept was not quite what it seemed to be. These seemingly subtle differences often turned out

to be significant enough to prevent straightforward transformation of DSL concepts to equivalents in embedded software.

Another difficulty was to verify the correctness of the implemented prototype library. The existing embedded software was used as the ground truth, but this ground truth was not necessarily the same as DSL models expressing what should be the same motion profile. This was apparent for the pin curing print mode evaluated in Section 7.4.2. Getting to the bottom of why library-based embedded software behaviour differed from existing embedded software took quite some doing. As no clear single source of truth was identified, we can consider both alternatives to be different versions of the truth.

> **RQ3**: What is the semantic gap between the existing DSL specification and the RTist-based embedded software?

While the existing SP DSL contains all concepts needed to describe carriage motion profiles, it turned out that ensuring that a model in this language can always be transformed to an embedded software compatible motion profile is not straightforward. To solve the issues that were encountered, an entirely new DSL was envisioned in the form of the Colorado Carriage Motion (CCM) language. Although this language was not implemented, the conceptual mapping was performed in Chapter 8. Although concepts from the SP DSL can be used to construct a prototype library that is integrated in the RTist embedded software, we can conclude the specification of the original SP DSL was not compatible with RTist model transformation. This is because certain things can be expressed in the DSL which cannot be expressed in the current embedded software, and vice versa.

> **RQ4**: Which tooling can be used to translate these JetBrains MPS models to RTist models?

This translation can be done using built-in MPS functionality, namely the generator aspect of MPS. As MPS models and RTist models are quite far apart conceptually, it is difficult to do this translation without in-between steps. As discussed in Section 4.2, two in-between steps were considered. The first is to use ESME [10]. The second is to define a library which can be generated from MPS, and is integrated in a RTist model. In this research we chose to use the latter, though the former is certainly an interesting topic for future research.

> **RQ5**: What is the efficacy of using Model Driven Engineering for carriage motion execution (in terms of reduced development time or improved functional correctness guarantees)?

This remains an open question for future work. Some comments can be made that provide some insight related to the question posed.

While developing the library, during every iteration a verification step using SIL was performed. Incremental compilation and running SIL for a single short print job took the largest part of an hour on an 8th generation Intel i7 laptop. If verification of the embedded software can be a separate task that only needs to be performed once, all other verification of different motion profiles can be done at DSL level. This reduces verification times from hours to seconds. In this regard, development time indeed may be reduced when generating motion profiles from the DSL rather than manually implementing each of them in embedded software.

The experimental results showed discrepancies between the DSL model representation of the considered pin cure motion strategy, and its implementation in embedded software. This likely an example of how difficult it is to keep different representations of the same concept in sync when there is no single source of truth.

To conclude, we were able to implement a prototype library for carriage motion strategies using DSL concepts, showing the feasibility of using model driven engineering to transform carriage motion DSL models to RTist-based embedded software.

Bibliography

[1] Gerald Czech, Michael Moser, and Josef Pichler. A systematic mapping study on best practices for domain-specific modeling. *Software Quality Journal*, 28(2):663–692, June 2020. doi:10.1007/s11219-019-09466-1. (Cited on page 6.)

[2] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2. (Cited on page 33.)

[3] HCL. Introducing HCL RTist [online]. 2022. URL: https://www.hcltech.com/software/rtist. (Cited on page 9.)

[4] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55. (Cited on page 33.)

[5] JetBrains. MPS Meta Programming System - Create your own domain-specific language [online]. 2022. URL: https://www.jetbrains.com/mps/. (Cited on pages 2 and 13.)

[6] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016. doi:10.3233/978-1-61499-649-1-87. (Cited on page 33.)

[7] Peter Liggesmeyer and Mario Trapp. Trends in embedded software engineering. *IEEE software*, 26(3):19–25, 2009. doi:10.1109/MS.2009.80. (Cited on page 2.)

[8] Apoorv Maheshwari, Navindran Davendralingam, Ali K. Raz, and Daniel A. DeLaurentis. Developing Model-Based Systems Engineering Artifacts for Legacy Systems. In *2018 AIAA Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics, 2018. URL: https://arc.aiaa.org/doi/abs/10.2514/6.2018-1213, doi:10.2514/6.2018-1213. (Cited on page 5.)

[9] Mattias Mohlin. Modeling Real-Time Applications in RTist [online]. 2020. URL: https://rtist.hcldoc.com/help/topic/com.ibm.xtools.rsarte.webdoc/pdf/RTist%20Concepts.pdf. (Cited on page 10.)

[10] Dmitrii Nikeshkin. *Domain-extensible model-driven embedded software development IDE.* PdEng report, Technische Universiteit Eindhoven, 2018. (Cited on pages 5, 20, and 44.)

[11] Atsushi Ohno, Takayuki Hikawa, Nobuhiko Nishio, and Takuya Azumi. Integration framework for legacy and generated code in mbd. In *Proc. WiP 26th Euromicro Conference on Real-Time Systems*, pages 9–12, 2014. (Cited on page 5.)

[12] Canon Production Printing. Colorado series [online]. 2022. URL: `https://cpp.canon/products/colorado-series/`. (Cited on pages 1 and 6.)

[13] 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*, pages 53–91. Springer, 2021. `doi:10.1007/978-3-030-73758-0_3`. (Cited on pages 5 and 13.)

[14] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. `doi:10.1038/s41592-019-0686-2`. (Cited on page 33.)

Glossary

I. List of Definitions

Term	Definition
Jetting	Depositing ink on the print medium.
Medium	The material on which the image is printed (often paper).
Engine	The physical machine that makes up the printer.
Printer	Different term for Engine.
Printing	The process of translating a digital image to a physical representation of that image on the print medium.
Result Image	Part of the medium where an image has appeared after a print job is completed.
Job	A task for the printer to print a complete image.
Carriage	A cart that moves along a beam on which printer components are mounted.
Print Head	Printer component that controls the deposition of ink on the print medium.
Swath	Longest continuous constant speed carriage movement where at least part of this movement is over the Result Image*.
Curing	Hardening or "drying" of ink deposited on the print medium.
Pin Curing	A specific type of curing that cures the ink before the ink has time to flow out.
Nozzle	Ink outlet from which ink is jetted.
Media Step	The movement of the print medium, perpendicular to the carriage movement.
Jerk	The rate of change in acceleration.
Slow Curing	A motion strategy where the cure carriage performs half the swaths of the print carriage.
Mono-Directional Printing	A motion strategy where the print carriage only performs swaths in a single direction.

*For actual footnote please see next page, as it did not fit here.

II. Acronyms

Term	Definition
API	Application Programming Interface
DSL	Domain Specific Language
CPP	Canon Production Printing
UML-RT	Unified Modeling Language for Real-Time
RTS	Run Time System
MPS	Meta Programming System
PSL	Print System Layout
SP	Scanning Productivity
CM	Carriage Motion
CCM	Colorado Carriage Motion
BoS	Begin of Swath
EoS	End of Swath
ESW	Embedded Software

*In the context of jetting this refers to the deposition of an image slice as ink on the print surface. To facilitate this deposition a specific carriage movement is needed. While there exists a relation between these concepts, in the context of carriage motion a swath is a more general concept. A swath by the print carriage always contains the section where ink is deposited, but may be longer than required for this ink deposition. The term is also used to denote movement of the cure carriage, which never deposits any ink (but instead 'deposits' light).

Appendix A

This appendix contains Scanning Productivity models of the motion strategies used in the evaluation in Section 7.4. In all cases a graphical representation of these models is followed by a textual representation. These models show the Situations corresponding to the begin and end Situations of swaths, which are used by the prototype library to calculate absolute BoS and EoS positions. These models are captured in Figure A3 through A8. More specifically:

- The default print mode from Section 7.4.1 is shown in Figure A3 and A4.
- The library pin cure print mode from Section 7.4.2 is shown in A5 and A6.
- The original software pin cure print mode from Section 7.4.3 is shown in Figure A7 and A8.

Lastly, a model is included that shows all Situations that represent a moment of interest for the considered pin curing motion strategy. In this model there is no clear relation between a Situation and a BoS or EoS position. This model is expressed in Figures A9, A10 and A11.

All models in this appendix use the same carriage layouts. To aid in linking the textual representation to the graphical depiction of Situations, Figure A1 and A2 show the carriage layouts annotated with the component names as used in the SP models that follow.

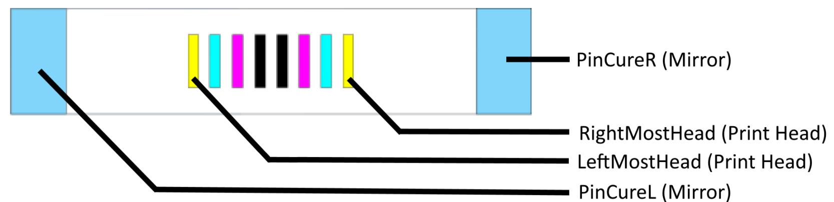

Figure A1: Colorado Print Carriage Layout, annotated with component names and component type between brackets.

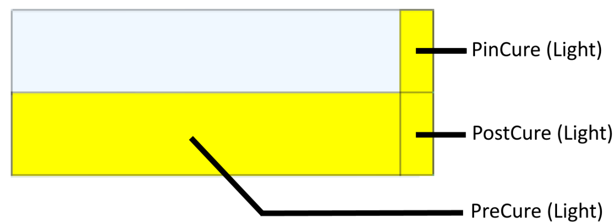

Figure A2: Colorado Cure Carriage Layout, annotated with component names and component type between brackets.

A final note that may aid in linking the experimental results from Section 7.4 to the models in this appendix; Situations that have “R2L” in their name correspond to even swaths, while Situations containing “L2R” correspond to odd swaths.

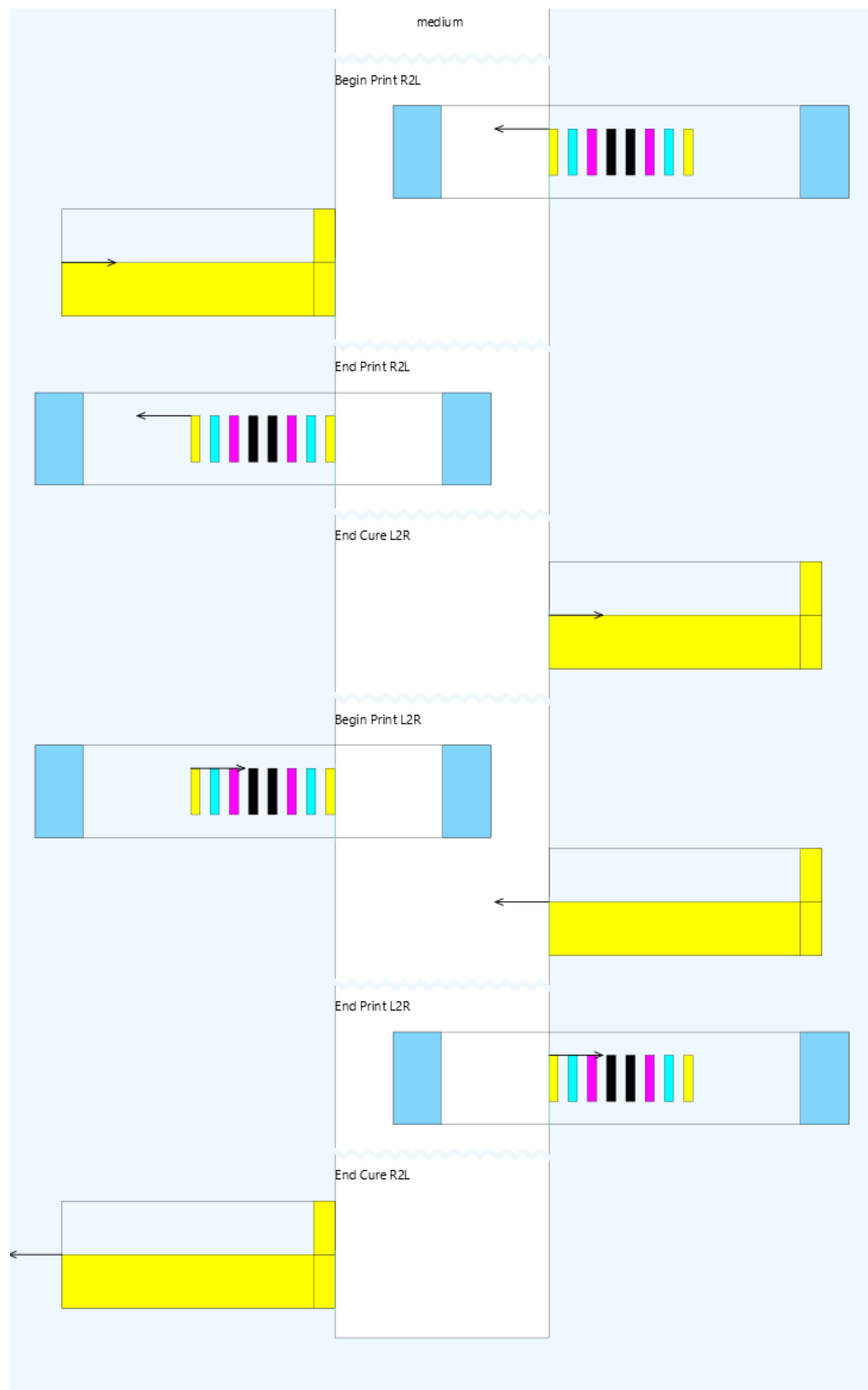

Figure A3: SP DSL model for default motion profile - BoS and EoS Situations - graphical representation.

CANON PRODUCTION PRINTING worksheet: Default_Library imports: Constants
Requirements

```
[1] repeated motion Default<sp_spec_curing>

  carriages: PrintCarriage CureCarriage
  medium: ResultImage

  situations:
    Begin Print R2L {
      (PrintCarriage) { left of LeftMostHead is aligned with right of ResultImage @ -printVelocity mm/s }
      (CureCarriage ) { right of PostCure is aligned with left of ResultImage @ cureVelocity mm/s }
    }
    End Print R2L {
      (PrintCarriage) { right of RightMostHead is aligned with left of ResultImage @ -printVelocity mm/s }
    }
    End Cure L2R {
      (CureCarriage) { left of PreCure is aligned with right of ResultImage @ cureVelocity mm/s }
    }
    Begin Print L2R {
      (PrintCarriage) { right of RightMostHead is aligned with left of ResultImage @ printVelocity mm/s }
      (CureCarriage ) { left of PreCure is aligned with right of ResultImage @ -cureVelocity mm/s }
    }
    End Print L2R {
      (PrintCarriage) { left of LeftMostHead is aligned with right of ResultImage @ printVelocity mm/s }
    }
    End Cure R2L {
      (CureCarriage) { right of PostCure is aligned with left of ResultImage @ -cureVelocity mm/s }
    }

  possible begin situations:
    << ... >>

  media steps:
    media step from End Cure L2R to Begin Print L2R
    media step from End Cure R2L to Begin Print R2L

  trajectories:
    constant speed for PrintCarriage : Begin Print R2L -> End Print R2L
    constant speed for PrintCarriage : Begin Print L2R -> End Print L2R
    constant speed for CureCarriage : Begin Print R2L -> End Cure L2R
    constant speed for CureCarriage : Begin Print L2R -> End Cure R2L

  timing constraints:
    << ... >>
```

Figure A4: SP DSL model for default motion profile - BoS and EoS Situations - textual representation.

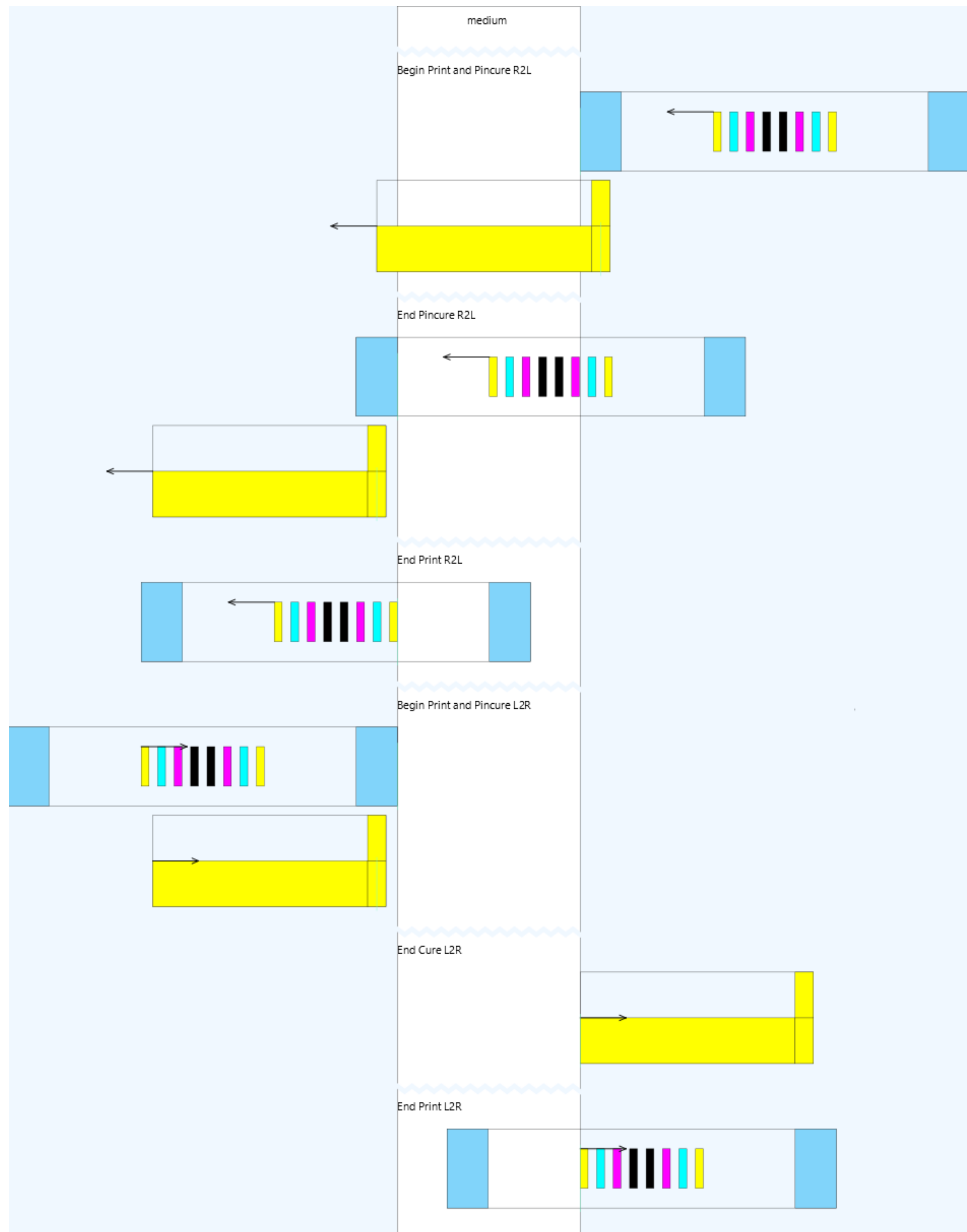

Figure A5: SP DSL model for pin cure motion profile - BoS and EoS Situations - graphical representation.

Canon
CANON PRODUCTION PRINTING

```
worksheet: LeadingPin_Library        imports: Constants
                                              Requirements

[1]
repeated motion LeadingPin<sp_spec_curing>

  carriages: PrintCarriage CureCarriage
  medium: ResultImage

  situations:
    Begin Print and Pincure R2L {
      (PrintCarriage) { left of PinCureL is aligned with right of ResultImage @ -printVelocity mm/s }
      (CureCarriage ) { center of PinCure is aligned with center of PinCureL  @ -printVelocity mm/s }
    }
    End Pincure R2L {
      (PrintCarriage) { right of PinCureL is aligned with left of ResultImage @ -printVelocity mm/s }
      (CureCarriage ) { center of PinCure is aligned with center of PinCureL  @ -printVelocity mm/s }
    }
    End Print R2L {
      (PrintCarriage) { right of RightMostHead is aligned with left of ResultImage @ -printVelocity mm/s }
    }
    Begin Print and Pincure L2R {
      (PrintCarriage) { right of PinCureR is aligned with left of ResultImage @ printVelocity mm/s }
      (CureCarriage ) { center of PinCure is aligned with center of PinCureR  @ printVelocity mm/s }
    }
    End Cure L2R {
      (CureCarriage) { left of PreCure is aligned with right of ResultImage @ printVelocity mm/s }
    }
    End Print L2R {
      (PrintCarriage) { left of LeftMostHead is aligned with right of ResultImage @ printVelocity mm/s }
    }

  possible begin situations:
    Begin Print and Pincure L2R known under alias ' L2R '
    Begin Print and Pincure R2L known under alias ' R2L '

  media steps:
    media step from End Print R2L to Begin Print and Pincure L2R
    media step from End Print L2R to Begin Print and Pincure R2L

  trajectories:
    constant speed for PrintCarriage : Begin Print and Pincure L2R -> End Print L2R
    constant speed for CureCarriage : Begin Print and Pincure L2R -> End Cure L2R
    constant speed for PrintCarriage : Begin Print and Pincure R2L -> End Pincure R2L
    constant speed for PrintCarriage : Begin Print and Pincure R2L -> End Print R2L
    constant speed for CureCarriage : Begin Print and Pincure R2L -> End Pincure R2L

  timing constraints:
    << ... >>
```

Figure A6: SP DSL model for pin cure motion profile - BoS and EoS Situations - textual representation.

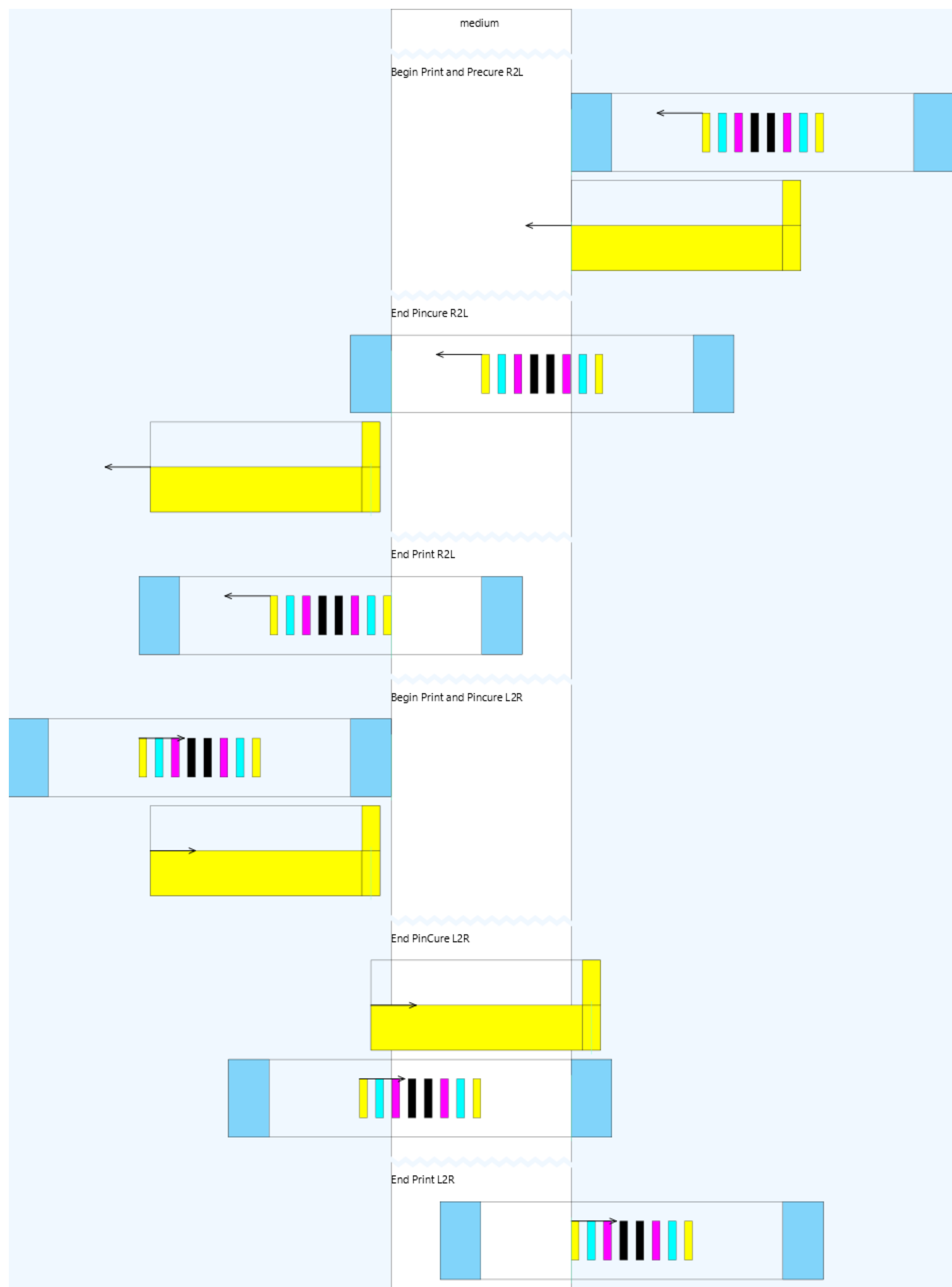

Figure A7: SP DSL model for original software pin cure motion profile - BoS and EoS Situations - graphical representation.

```
Canon
CANON PRODUCTION PRINTING    worksheet: LeadingPin_Library_Copy_Original    imports: Constants
                                                                                      Requirements

[1] repeated motion LeadingPin<sp_spec_curing>

      carriages: PrintCarriage CureCarriage
      medium: ResultImage

      situations:
        Begin Print and Precure R2L {
          (PrintCarriage) { left of PinCureL is aligned with right of ResultImage @ -printVelocity mm/s }
          (CureCarriage ) { left of PreCure is aligned with right of ResultImage  @ -printVelocity mm/s }
        }
        End Pincure R2L {
          (PrintCarriage) { right of PinCureL is aligned with left of ResultImage @ -printVelocity mm/s }
          (CureCarriage ) { center of PinCure is aligned with center of PinCureL  @ -printVelocity mm/s }
        }
        End Print R2L {
          (PrintCarriage) { right of RightMostHead is aligned with left of ResultImage @ -printVelocity mm/s }
        }
        Begin Print and Pincure L2R {
          (PrintCarriage) { right of PinCureR is aligned with left of ResultImage @ printVelocity mm/s }
          (CureCarriage ) { center of PinCure is aligned with center of PinCureR  @ printVelocity mm/s }
        }
        End PinCure L2R {
          (CureCarriage ) { center of PinCure is aligned with center of PinCureR  @ printVelocity mm/s }
          (PrintCarriage) { left of PinCureR is aligned with right of ResultImage @ printVelocity mm/s }
        }
        End Print L2R {
          (PrintCarriage) { left of LeftMostHead is aligned with right of ResultImage @ printVelocity mm/s }
        }

      possible begin situations:
        Begin Print and Pincure L2R known under alias ' L2R '
        Begin Print and Precure R2L known under alias ' R2L '

      media steps:
        media step from End Print R2L to Begin Print and Pincure L2R
        media step from End Print L2R to Begin Print and Precure R2L

      trajectories:
        constant speed for PrintCarriage : Begin Print and Pincure L2R -> End Print L2R
        constant speed for PrintCarriage : Begin Print and Pincure L2R -> End PinCure L2R
        constant speed for CureCarriage : Begin Print and Pincure L2R -> End PinCure L2R
        constant speed for PrintCarriage : Begin Print and Precure R2L -> End Pincure R2L
        constant speed for PrintCarriage : Begin Print and Precure R2L -> End Print R2L
        constant speed for CureCarriage : Begin Print and Precure R2L -> End Pincure R2L

      timing constraints:
        << ... >>
```

Figure A8: SP DSL model for original software pin cure motion profile - textual representation.

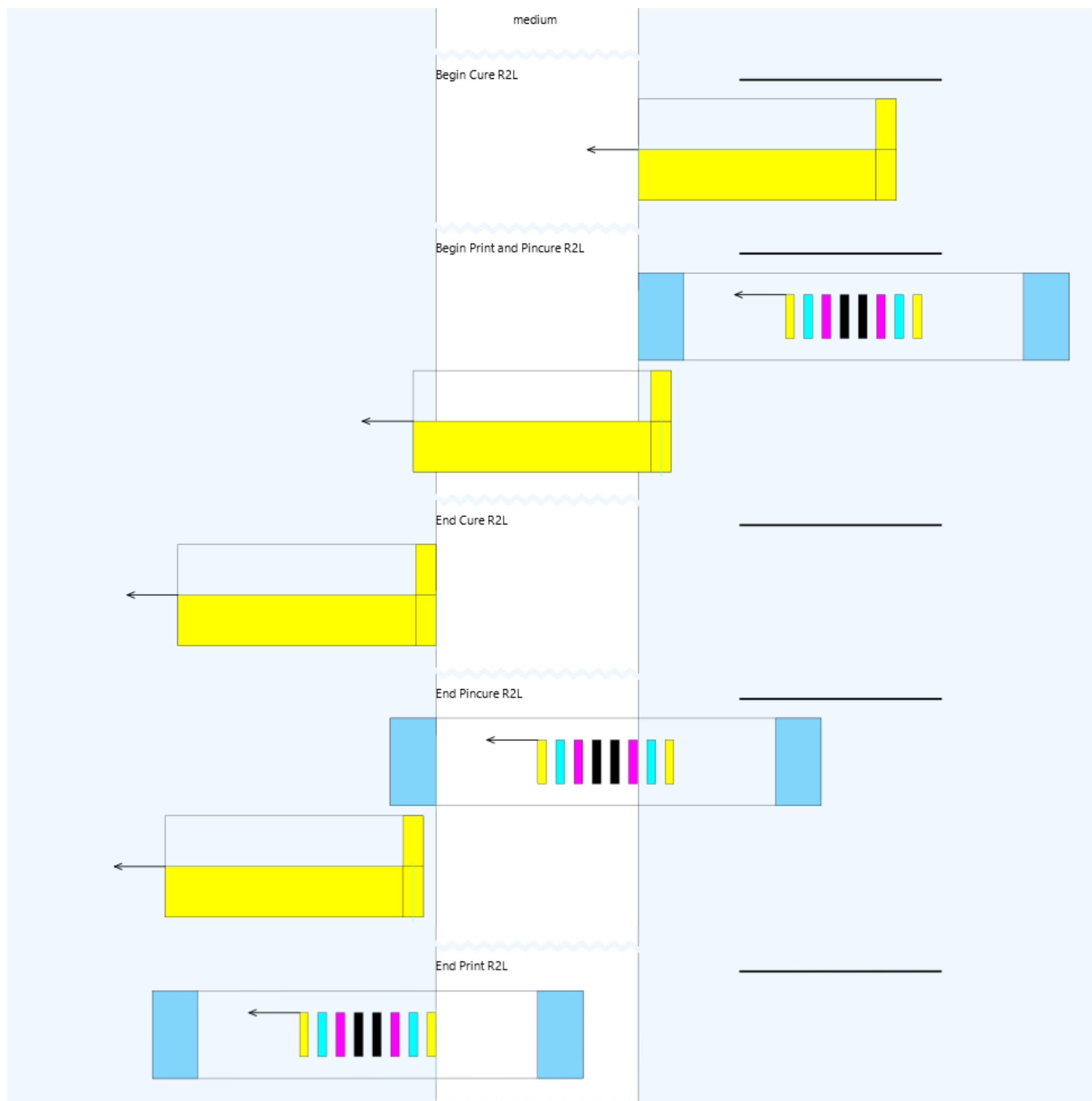

Figure A9: SP DSL model for pin cure motion profile - right-to-left Situations - graphical representation.

Figure A10: SP DSL model for pin cure motion profile - left-to-right Situations - graphical representation.

```
Canon
CANON PRODUCTION PRINTING    worksheet: LeadingPin_Original        imports: <- Constants
                                                                            <- Requirements

[1] repeated motion LeadingPin<sp_spec_curing>

      carriages: PrintCarriage CureCarriage
      medium: ResultImage

      situations:
        Begin Cure R2L {
          (CureCarriage) { left of PreCure is aligned with right of ResultImage @ -printVelocity mm/s }
        }
        Begin Print and Pincure R2L {
          (PrintCarriage) { left of PinCureL is aligned with right of ResultImage @ -printVelocity mm/s }
          (CureCarriage ) { center of PinCure is aligned with center of PinCureL @ -printVelocity mm/s }
        }
        End Cure R2L {
          (CureCarriage) { right of PostCure is aligned with left of ResultImage @ -printVelocity mm/s }
        }
        End Pincure R2L {
          (PrintCarriage) { right of PinCureL is aligned with left of ResultImage @ -printVelocity mm/s }
          (CureCarriage ) { center of PinCure is aligned with center of PinCureL @ -printVelocity mm/s }
        }
        End Print R2L {
          (PrintCarriage) { right of RightMostHead is aligned with left of ResultImage @ -printVelocity mm/s }
        }
        Begin Print and Pincure L2R {
          (PrintCarriage) { right of PinCureR is aligned with left of ResultImage @ printVelocity mm/s }
          (CureCarriage ) { center of PinCure is aligned with center of PinCureR @ printVelocity mm/s }
        }
        Begin Cure L2R {
          (CureCarriage) { right of PostCure is aligned with left of ResultImage @ printVelocity mm/s }
        }
        End Pincure L2R {
          (PrintCarriage) { left of PinCureR is aligned with right of ResultImage @ printVelocity mm/s }
          (CureCarriage ) { center of PinCure is aligned with center of PinCureR @ printVelocity mm/s }
        }
        End Print L2R {
          (PrintCarriage) { left of LeftMostHead is aligned with right of ResultImage @ printVelocity mm/s }
        }
        End Cure L2R {
          (CureCarriage) { left of PreCure is aligned with right of ResultImage @ printVelocity mm/s }
        }

      possible begin situations:
        Begin Print and Pincure L2R known under alias ' L2R '
        Begin Print and Pincure R2L known under alias ' R2L '

      media steps:
        media step from End Print R2L to Begin Print and Pincure L2R
        media step from End Print L2R to Begin Print and Pincure R2L

      trajectories:
        constant speed for PrintCarriage : Begin Print and Pincure L2R -> End Print L2R
        constant speed for PrintCarriage : Begin Print and Pincure R2L -> End Pincure R2L
        constant speed for PrintCarriage : Begin Print and Pincure R2L -> End Print R2L
        constant speed for PrintCarriage : Begin Print and Pincure L2R -> End Pincure L2R
        constant speed for CureCarriage : Begin Print and Pincure L2R -> End Pincure L2R
        constant speed for CureCarriage : Begin Cure L2R -> End Cure L2R
        constant speed for CureCarriage : Begin Print and Pincure R2L -> End Pincure R2L
        constant speed for CureCarriage : Begin Cure R2L -> End Pincure R2L
        constant speed for CureCarriage : Begin Cure R2L -> End Cure R2L

      timing constraints:
        << ... >>
```

Figure A11: SP DSL model for pin cure motion profile - all Situations - textual representation.