JetBrains MPS as Core DSL Technology for Developing Professional Digital Printers

Eugen Schindler, Hristina Moneva, Joost van Pinxten, Louis van Gool, Bart van der Meulen, Niko Stotz, and Bart Theelen

Abstract To address the challenges of efficiently performing continuous innovation with sustainable quality, Canon Production Printing envisions the exploitation of models during the complete lifecycle of printer variants. The design of professional digital printers involves several engineering disciplines ranging from software and electrical and mechanical hardware to physics and chemistry. All these engineering disciplines already exploit various models in diverse tools to support their design activities. Apart from exploiting commodity modeling tools that are used in domain-specific ways, specialized domain-specific modeling tools are also developed. At the time of writing, Canon Production Printing had selected JetBrains MPS as one of the core technologies to interconnect the diverse range of models at the specification level. Such models are, for example, used for configuring virtual printers for analysis (e.g., by means of simulation), generation of documentation, and automating synthesis in some engineering disciplines. To allow for automated processing, MPS is also used for capturing domain-specific knowledge in models that has not yet been formalized. At the time of writing, approximately 10 people at Canon Production Printing develop domain-specific languages using MPS and approximately 40 people use these as part of their daily printer development activities. This chapter gives an overview of some of the applications of MPS at Canon Production Printing and how it is changing the way-of-working to achieve efficient continuous innovation with sustainable quality.

E. Schindler · H. Moneva · J. van Pinxten (✉) · L. van Gool · B. van der Meulen · N. Stotz · B. Theelen
Canon Production Printing Netherlands B.V., Venlo, The Netherlands
e-mail: eugen.schindler@cpp.canon; hristina.moneva@cpp.canon; joost.vanpinxten@cpp.canon; louis.vangool@cpp.canon; bart.vandermeulen@cpp.canon; niko.stotz@cpp.canon; bart.theelen@cpp.canon

A. Bucchiarone et al. (eds.), *Domain-Specific Languages in Practice*,
https://doi.org/10.1007/978-3-030-73758-0_3

1 Introduction

Canon Production Printing develops high-end professional digital printers in three categories: (1) industry-leading continuous-feed printers for massive print volumes and fast, high-quality results in full color or black & white; (2) highly efficient, high-volume printers for in-house printing or publishing; and (3) large-format printers for stunning display graphics and high-quality CAD/GIS applications. These products serve the professional print market with suitable trade-offs between multiple system Key Performance Indicators (KPIs) such as productivity, perceived image quality (i.e., how good does the printed image look), print robustness (e.g., how well does the ink stick to the medium), and cost. Figure 1 highlights some example printer families with an indication of their productivity capabilities and physical sizes.

The printers in Fig. 1 exploit an inkjet print process, which covers steps such as image processing, positioning the jetted ink on media, and spreading & solidification of the ink. How each print process step is realized has an impact on the various system KPIs. A multitude of engineering disciplines (software, electrical and mechanical hardware, physics, and chemistry) is involved in the development of print processes and printers, and hence in realizing competitive values for all the system KPIs.

Development within Canon Production Printing exploits several model-based approaches in all engineering disciplines to cope with the ever-increasing complexity. The complexity of professional digital printers is nowadays fairly comparable to that of high-end cars. At Canon Production Printing, model-based approaches have already proven to be essential for efficiently performing continuous innovation with sustainable quality in a highly competitive market.

Fig. 1 Examples of professional digital printers developed by Canon Production Printing

An important ingredient of model-based development at Canon Production Printing is to have good tool support to do the actual modeling. Architects, designers, and engineers often rely on tools that are to some extent targeted for addressing specific aspects of a printer. For example, the mechanical design relies on using a CAD tool, which is, however, an unfamiliar development environment for a software designer. Nevertheless, the mechanical and software designer are working on the same printer, and hence part of the knowledge captured in their models must be the same to ensure that together they come to a design that will actually work. Exposing (the knowledge captured in) models to a wider audience than the original context from which they originated is an essential ingredient of achieving consistent designs within and across engineering disciplines. Instead of teaching the wider audience to use the tool in which a model was originally created, it can be more appropriate to introduce a (meta)tool that allows for automated exchange of the knowledge captured in various domain-specific models (which are developed in a diversity of tools). This allows for much better reuse of models within and across engineering disciplines. In summary, the benefits of using a (meta)tool include:

- Users don't have to learn the quirks of numerous tools.
- Within one (meta)tool, multiple models can be integrated much more intimately.
- Integrating and maintaining one (meta)tool in development (and automated build) environments is less work than integrating and maintaining multiple tools.

MPS is a core (meta)tool that is used extensively within Canon Production Printing. However, it is not only used for realizing consistent exchange of knowledge captured in models developed in more domain-specific tools (such as the CAD tool example). It is also being used as the main tool for creating models to capture knowledge that has not yet been formalized. This allows computer-based processing for more automated printer development as essential extensions to the capabilities of consistent exchange of knowledge between domain-specific tools. This chapter highlights several modeling efforts that emerged within Canon Production Printing and how MPS played a role in these efforts. Figure 2 gives a high-level impression

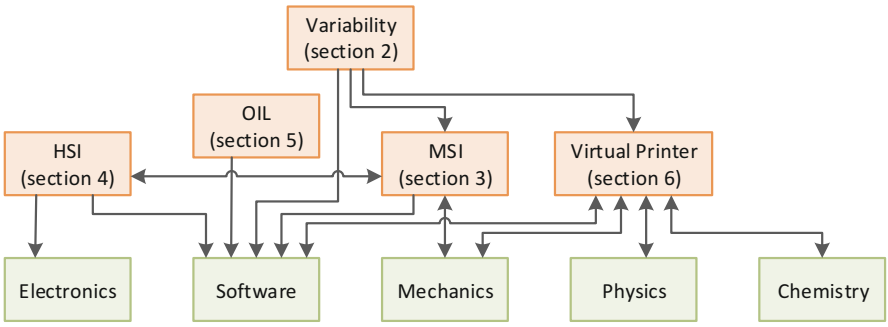


Fig. 2 Relations between the DSLs in this chapter and domain models in various engineering disciplines. In reality, several additional DSLs are involved to cover some of the expressed relations

of the main relations between the different DSLs discussed in this chapter and knowledge captured in domain models for the involved engineering disciplines.

Section 2 describes how MPS is deployed for coping with the variability that is the result of supporting many product variations within a single piece of software. Section 3 shows how knowledge about mechanical properties previously encoded in C++ was lifted to models, enabling effective and consistent communication between different stakeholders. Section 4 treats a similar abstraction effort in the context of hardware-software interfaces. Where Sect. 2 shows how we used and slightly extended an existing "horizontal" DSL to address a specific aspect of our software development, Sect. 5 presents a "horizontal" DSL that was entirely developed at Canon Production Printing for the description of control behavior. In contrast to these two "horizontal" languages, Sect. 6 presents a real "vertical" (print domain-specific) effort to model several aspects of our print process. Section 7 discusses some meta-level efforts that should help us to make the power of domain-specific modeling available to a larger user community. Finally, Sect. 8 summarizes some lessons learned from applying modeling, and in particular MPS.

2 Product Variability

Canon Production Printing encounters variations in different contexts: between products, within shared platforms, by used or supported hardware, or depending on settings or purchased licenses. Historically, these variations are maintained on an ad-hoc basis within our software. This has typically led to locally optimal solutions to determine which features are supported for a specific configuration. However, this evolution also led to issues such as different information sources in different modules for the same variation point, an incomplete overview of applicable variations and their dependencies, and higher testing and maintenance effort.

We started to migrate our variations from various contexts to an MPS-based feature model, based on the sound theory of Product Line Engineering [2, 16, 17]. The feature model language was implemented by *itemis* as a result of the IETS3 project [14]. *itemis* also provided a complementary extension language to support our specific use case with named constraints and feature flags. We further leveraged MPS's language extensibility by attaching additional validations to the feature model language, and implementing a generator to target our specific run-time environment.

Some of our ad-hoc variations (expressed as conditions in regular source code) have already been successfully migrated to MPS-based feature models. The source code artifacts generated from these feature models have also been deployed to production. At the time of writing, these feature models contain approximately 70 features, 30 attributes, and 30 constraints. We expect these numbers to grow by at least one order of magnitude. We indeed observe the expected benefits to our code structure, such as less code duplication, single source of information,

and clearer dependencies between variations. Hence, we intend to continue the migration process.

Currently, our feature model comprises (rather low-level) technical features expressing variability from a technical implementation perspective, which we can therefore migrate directly to implementation code, for example. Due to their technical nature, these models are used by developers. In the future, we want to explore combining different technically minded feature models, or relate them to more abstract domain-focused feature models. Such domain-focused feature models express variability from the customer perspective when buying or using our printer systems.

Introducing MPS to our software developers also came with some hurdles. A first hurdle was usability-related: especially the unconventional editing experience and the abundant warnings required a change in mindset. Second, we missed editor-related features such as triggering transitive model updates and amending the inspector view of existing concepts for our extension language. A final hurdle relates to our generation target being a programming language that is not yet available through a DSL model within MPS. Our solution was to exploit the external extension PlaintextGen [8] to generate plain text instead of using a model-to-model transformation.

2.1 Variability at Canon Production Printing

Within Canon Production Printing, we handle variations in different contexts and at different points in the product lifecycle. Some of these variations only concern our own development, while others are visible to the market or to a specific customer. Examples of internal variations include the specific hardware components within a printer and products based on the same platform. To the market, variations might be different products or versions of the same product family, supported so-called finishing equipment (e.g., third-party staplers or stackers), and available optional features. Specific customers can vary in their purchased licenses, product environment, and individual printer settings. We need to consider platform-related variations very early on when designing a shared platform. Variations between printer products, supported equipment, or optional features are part of product design, but might change with subsequent versions of the same product. Variations based on installed finishers, purchased licenses, or individual settings are only known at run-time.

2.1.1 Designing for Variability

Handling variations ad-hoc leads to a range of design challenges. It is not easily visible which variation can occur in which combination, and whether we covered

all relevant combinations. We might have variations without any actual difference, leading to code duplication and higher effort for testing and issue fixing.

Many variations depend on each other: a specific kind of ink might be available only on some product releases, and can only be used if the customer purchased the appropriate license. Tracking such dependencies is infeasible if they are only expressed as part of the software source code.

At each variation point within the software, we have to decide whether some variation is valid or not (think of "Is glossy media[1] supported?" as an example). In one software module (e.g., built-in UI of the printer), we might decide depending on the installed hardware revision. In another module (e.g., printer driver), the same decision might be based on an internal code name, because the code name changes with the hardware revision, and reading the revision directly might be cumbersome. Assume for a new product that the installed hardware revision stays the same, but the internal code name changes. This leads to potential discrepancies: the former module decides the variation is valid (based on the hardware revision), but the latter module decides the same variation is invalid (based on the internal code name). Thus, the user could select glossy media directly on the printer, but not in the printer driver.

2.1.2 Development Effort and User Base

The internal extensions to the Variability language are supported by one language engineer. The same engineer develops the run-time environment. At the time of writing, the total effort amounted to less than 10 weeks. Most of the language development is tasked to *itemis*. We extended their languages with approximately 20 concepts, checking rules, and intentions. The generator toward our run-time environment constituted the bulk of in-house development.

At the time of writing, the language is used by approximately one dozen users. In total, we spent roughly 1 week on training; general MPS education took the largest share of this effort. We expect the majority of our software engineers to use this tool chain.

2.2 Product Line Engineering

The problem of variations has been thoroughly researched in Computer Science under the term *Product Line Engineering* [2, 16, 17]. Fundamentally, they describe *features* as boolean variables (i.e., each feature might be enabled or disabled); at each *variation point*, we query the appropriate feature and act accordingly. The dependencies between features are described with *constraints*. Product Line

[1] A type of paper .

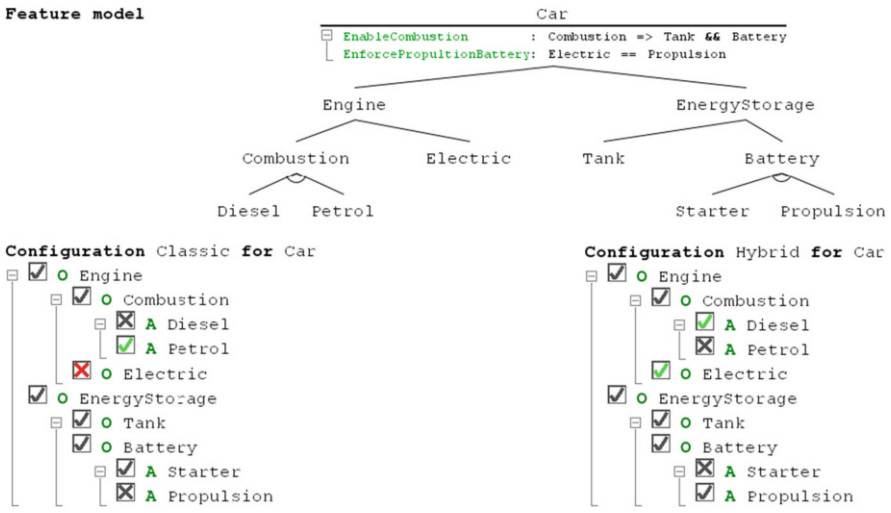


Fig. 3 A *feature model* and two *configurations* in MPS (see footnote 2) Each node in the tree represents a *feature*. Lines represent *constraints* between features. Additional *constraints* are expressed in predicate logic. The configurations denote *enabled* features with a check-mark and *disabled* features with a cross

Engineering research came up with an easy-to-understand, tree-based notation for the most common constraints; more complex constraints are expressed with predicate logic. A *feature model* contains all features and their constraints. One set of values, i.e., enabled or disabled state for each feature in a feature model, is called a *configuration*. A configuration is *valid* if it fulfills all constraints. Figure 3 shows a simple example.[2]

2.2.1 Solution Outline

We aim at migrating any ad-hoc variation into unique features. All variation points that relate to the same variation should query the same feature; this ensures consistent effects of enabled or disabled features throughout the system. Each variation point should query as few features as possible; any kind of dependencies on other features should be modeled as constraints. This removes all dependency logic from the code, and ensures that the same source of information is used.

To cater for our needs, we augment the feature model with feature-specific *flags*. These allow extra information to be captured, such as *derived*, *plain*, or *run-time*. The value of a *derived* feature is determined by constraints: only this

[2]We use a car-based example here because of the more generally known concepts, in contrast to lesser known concept variations in the printing domain.

amendment allows constraints to actively enforce dependencies between features. A *run-time* feature can change its value at run-time - contrary to *plain* features that are configured at design-time. This allows us to model features such as installed finishers or printer-specific settings.

We need to access the feature model at run-time to query for the relevant feature at each variation point. Resource limitations prohibit us from deploying MPS on a printer. Furthermore, accessing MPS models from the existing system would require considerable additional engineering effort. Thus, we generate source code from the feature model. The generated code holds all the knowledge from the feature model (i.e. all defined features, constraints, and attributes), and ties in with our manually implemented run-time framework. In combination, this code ensures that only features flagged *run-time* can be changed on the printer, and all features flagged *derived* are updated accordingly. This means that we reuse the same constraints to check for valid configurations at design-time and run-time.

2.2.2 Example

Assume the legacy code contains the pseudo-code fragments shown in Listings 1 and 2 for the UI module and job control module respectively. At both conditional expressions (i.e., the variation points), we have to make a decision based on product variant, available hardware revision, and whether glossy media (see footnote 1) is licensed. `productCode` is set based on product variant and hardware revision. The code exhibits several of the listed issues: different sources of information for the same decision, difficult to correlate both variation points, hard-coded dependencies to other variations.

Listing 1 Legacy UI

```
if (productCode == "nextGenPrinter" and isLicensed(GlossyMedia)) {
    glossyMediaCheckbox.visible = true
} else {
    glossyMediaCheckbox.visible = false
}
```

Listing 2 Legacy job control

```
if (job.media.type == Glossy) {
    if (productVariant != Versatile
        or hardwareRevision < 2
        or not isLicensed(GlossyMedia)) {
        refuseJob("glossy not supported or not licensed")
    }
}
```

After introducing the feature model in Fig. 4, we can replace both conditions by a simple query for feature `MediaGlossy`.

2.3 Advantages of MPS

Our feature model is implemented in MPS, based on a generic feature modeling language provided by *itemis*, who integrated a constraint solver with their feature model language. This constraint solver provides instant feedback on inconsistent models or contradicting constraints within the model. On our request, *itemis* implemented an accompanying language to support the feature flags. We further leveraged MPS's language extensibility to add custom validations on top of the existing feature model language. We exploit this, for example, to ensure unique feature names and that feature names adhere to the rules of the target language. As MPS allows additional generators without changing the source model's language, we have implemented our own generator based on the language stack developed by *itemis*.

The MPS projectional editor integrates seamlessly both the graphical feature tree and the textual constraint expressions. Modeling and reasoning about all features and their dependencies is quite challenging. However, the integrated rendering and seamless editing allow the focus to be placed on the inherent complexity, rather than dealing with inadequate or split up editors (i.e., tooling issues).

2.4 Shortcomings of MPS

As we are replacing the logic with variation points throughout our code base, team members regularly encounter MPS for the first time when they adopt our feature model. Being developers, this user group is not discouraged by MPS's IDE

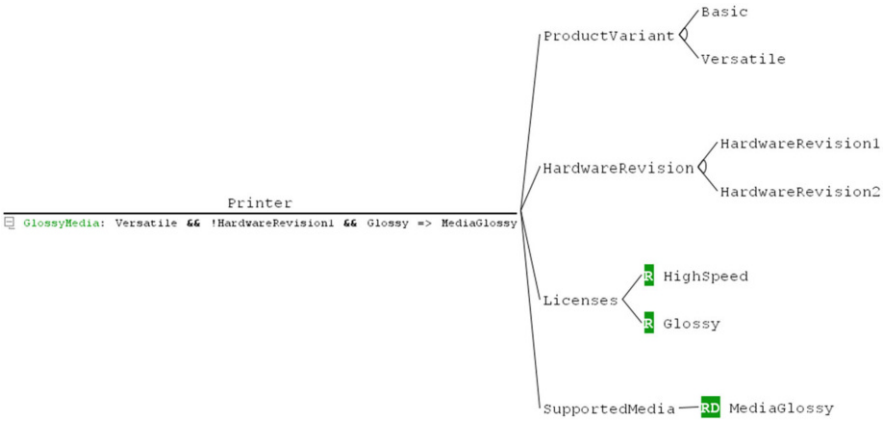


Fig. 4 Example feature model in MPS deriving the supported media from design-time features (product variant and hardware revision) and run-time feature (license) *R* and *D* on dark background represent the feature-specific flags Runtime and Derived

appearance. However, they still have difficulties in working with the tool without close guidance. The main usability issues include the separation between context menu and intention menu, the unfamiliar keyboard navigation and selection scheme.

The feature modeling language consists of two main aspects: the feature model and configurations. After every change to the feature model, the user has to explicitly adapt each configuration to the changes by triggering an intention. This could be avoided if MPS would provide language developers with an easy mechanism to propagate changes to other parts of the model, while keeping manual changes on the propagation target untouched. MPS supports generic language extension through *node annotations* and accompanying amendments to the projected main editor. However, a language extension cannot easily amend the projected inspector editor.

The target programming language of our custom generator is not available as MPS language, and is too complex to easily implement. MPS's TextGen is not well suited to generate large amounts of text, especially if the generated structure differs from the input model. We resolved this issue by leveraging the PlaintextGen extension [8].

2.5 Status and Outlook for Product Variability Modeling

At the time of writing, we migrated approximately one dozen software variation points to the feature model and successfully delivered products based on this development into production. We are continuing the migration of ad-hoc variation points to the feature-based approach. In this process, we extend the internal user base of MPS-based modeling. Several other areas, including hardware variations, are being prepared to leverage feature modeling in MPS (see also next sections).

In a first step, the feature models on the software and hardware side are expected to stay independent. Combining these models might lead to additional insights on dependencies and reuse-benefits. We intend to investigate this direction in the future.

As we migrate source code to feature models, these models are very detailed and technical. However, most features stem from domain, product, or business variations (similar to what is described in Sect. 2.1). It would be interesting to model these more abstract features and relate them to the technical feature models.

3 Mechanics-Software Interface

The Mechanics-Software Interface (MSI) is a collection of DSLs that are used together to describe how sheets of paper are transported through the paper path of a printer. The *paper path* is a mechanical system described using three-dimensional components that exist in the mechanical sub-domain of printers, such as *transport*

belts, metal *sheet guides*, (movement) sensors, and actuators such as *pinches* and *deflectors*.

Canon Production Printing developed the MSI DSLs to support the HappyFlow model-based design approach for sheet movement in so-called cut-sheet printers. This *HappyFlow* approach is "*the conscious simplification of the model, where only the desired behavior of a sheet and the ideal movements of all parts are modeled*" (for more details, refer to [12]). HappyFlow decouples the low-level (non-ideal) physical behavior from the high-level scheduling of the sheets. The scheduling defines for each sheet when it is released and when it comes back for the second (duplex) printing. Scheduling is not trivial due to the large return loop and the run-time media variations and corresponding constraints between subsequent sheets. The *HappyFlow* approach allows reasoning regarding the effectiveness of a paper path to transport individual sheets as well as the impact of timing requirements on streams of sheets. Such timing requirements are imposed so that certain actions can be performed on a sheet. Examples of these actions include deflecting a sheet,[3] fixating the image onto a sheet using heaters, and subsequently cooling the sheet.

This section describes part of the evolution of the *HappyFlow* model-based design approach since its initial publication in [12] and what role MPS played in adopting a continuous-integration approach. Since the original *HappyFlow* approach was introduced, paper paths have become significantly larger and more complex. The complexity of the sheet transport for the printing domain requires a broad understanding of physical properties, as well as an in-depth understanding of scheduling and interaction of software components. It is therefore beneficial to alleviate the mental burden of the paper path designers by capturing reusable domain knowledge. Capturing this knowledge also allows automatic generation of software artifacts, as well as automated verification and optimization techniques.

This section presents how MPS has been used to capture the cut-sheet printer-specific mechanics and software domains and how it interfaces between these domains. We specifically highlight how MPS helped to achieve or improve:

- **Modularity** of the paper path in relation to product variability (see also Sect. 2)
- **Communication** between interdisciplinary teams
- **Continuous Integration** of model-based design artifacts

Figure 5 illustrates how the paper path (mechanical CAD models), parts, sheet timing, and sheet scheduling constraints are connected in MPS and how the generated software artifacts relate to each other. We use this overview throughout the remainder of this section.

[3] For example, a sheet needs to enter the duplex loop only if it needs to be printed again.

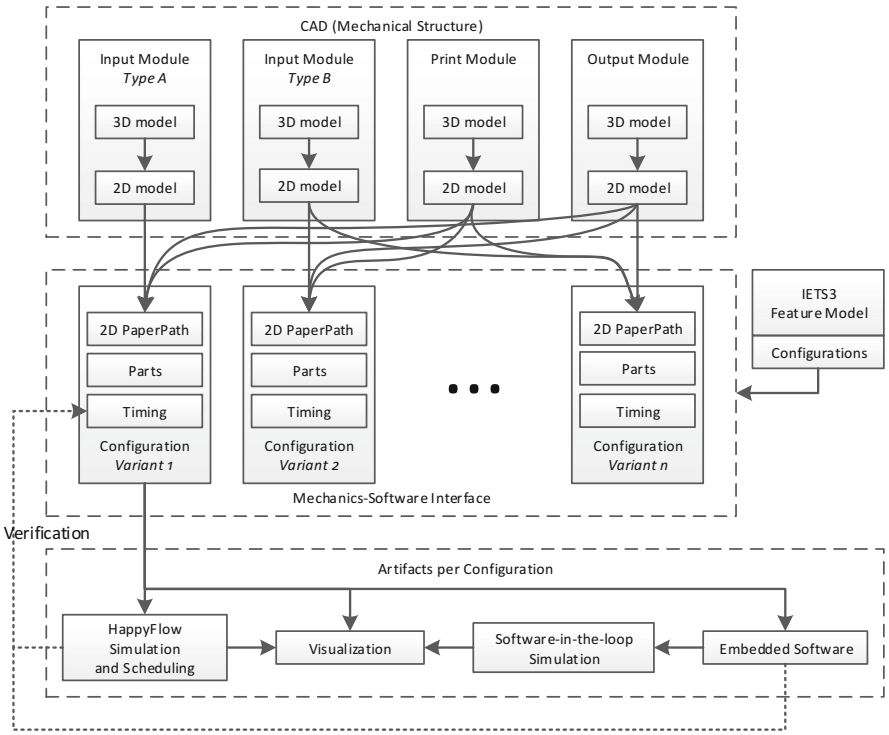


Fig. 5 Example models and artifacts in MSI

3.1 Paper Paths

The three-dimensional paper path layout is constructed in a CAD environment. A two-dimensional definition of the paper path is exported as an XML file, which is in turn imported by MPS. The 2D paper path model is most naturally displayed as a graphical model (see also Fig. 6). This allows easy communication between the mechanical engineers and the engineers defining timing requirements. The *segments* (lines) contain annotations on *points-of-interest* (POIs), which refer to locations of, e.g., sensors, pinches, and synchronization points. Different *routes* are defined based on these segments for sheets that need to be printed once (simplex) or twice (duplex). Each route describes which POIs are encountered during transportation of a sheet.

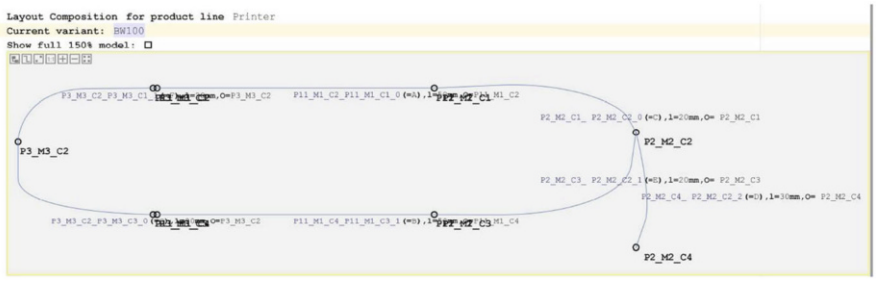


Fig. 6 Example paper path layout in MPS consisting of multiple *segments* between *points-of-interest* (the small circles). Such a paper path layout is a composition of paper path building blocks (such as input, process, and output) that are instantiated (and connected at points-of-interest that allow such connections) as a result of an instance of the product variability model explained in Sect. 2

3.2 Parts

The software that reads the sensors of the paper path and controls its actuators requires information about the physical properties of the motors (type, torque, maximum acceleration, or velocity), sensors, and pinches (circumference), for example specifying whether a motor in the paper path is realized through a stepper-motor or a brush-less direct-current motor. In addition, these motors are connected through different gears to one or multiple pinches or belts in the paper path.

This mechanical information is not provided in the CAD export, and is described by means of *parts* in the MSI DSLs. A part typically has multiple stakeholders in different engineering disciplines and, as such, different representations. The most detailed representation contains all required and optional attributes of a part, whereas the high-level overview of logical connections between motors, gears, and pinches can be shown best as a graphical diagram. MPS allows us to switch effortlessly between the different representations, which eases communication and verification.

3.3 HappyFlow Sheet Timing

The sheet timing model defines at what speed and acceleration a sheet should ideally move at any position in a route. This has traditionally been engineered by coding such behavior directly in C++. The sheet timing DSL started out as a C++ API, which meant that a significant amount of detail needed to be programmed, which is not directly relevant to the conceptual definition of the sheet timing. Lifting the C++ API to a proper DSL in MPS means that the designer responsible for sheet timing can focus on the definition of this timing behavior without being distracted by any C++ peculiarities. The timing-related instructions are typically physical

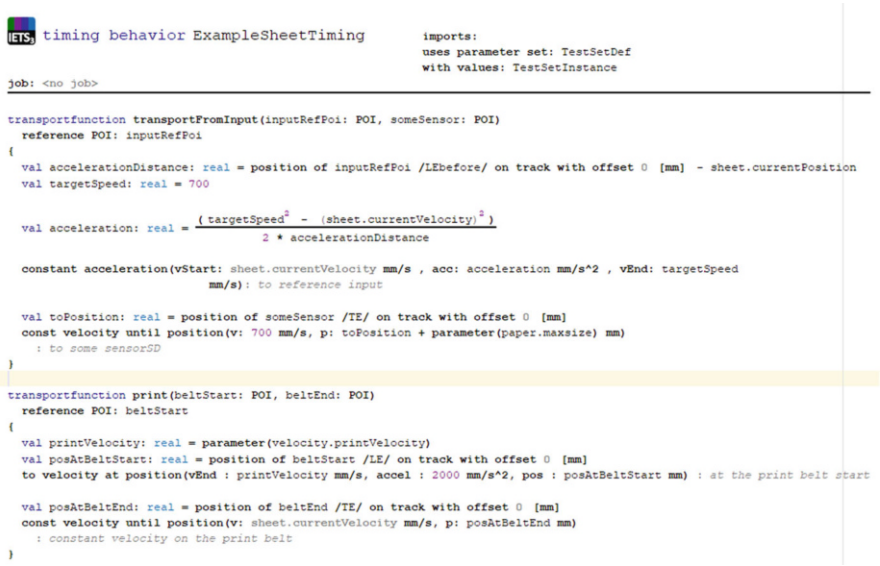


Fig. 7 Example functionality used in the sheet timing model in MPS. The focus is on the sheet's actions and mathematical computations, with connections to its movement through the printer

computations regarding the velocities and accelerations of the sheet. For such computations in our domain, we have used the general-purpose functional kernel called KernelF [20]. Integrating KernelF through MPS's language composition into our own DSL constructs immediately enabled sufficient support for computational expressions used in the domain. The result is that the sheet timing model is more easily captured using general statements and mathematical notation instead of C++. Figure 7 gives an example of the succinctness of the sheet timing model. These sheet timing models are used to generate the information on where the sheet should be at what time. Figure 8 shows an example inline graph with the position (i.e., displacement in the sheet track) of a sheet versus time for two sheets, one simplex, one duplex.

A benefit of using MPS for the sheet timing definitions is that particular details can be hidden, depending on the level of detail that is required to present the functionality to involved stakeholders. Such information can be hidden either in the MPS Inspector window or in different projections of the same model. This greatly enhances the conciseness and readability of certain parts of the sheet timing calculation.

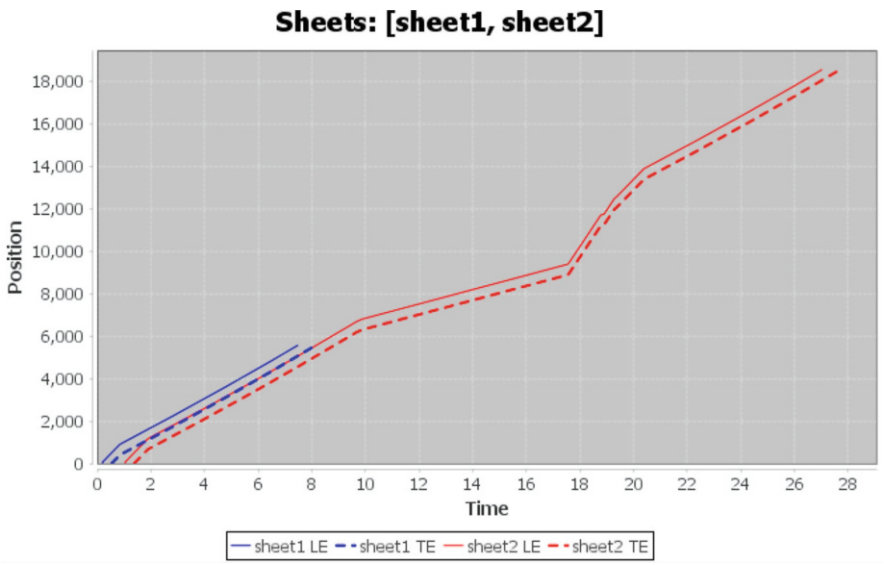


Fig. 8 Sheet position (displacement) vs time diagram

3.4 Sheet Scheduling Constraints and Verification

The sheet timing model imposes constraints on the transportation of each individual sheet. In this section, we describe how we deal with the scheduling constraints that impose constraints between subsequently transported sheets. The value of these constraints depends on the media properties and functional requirements to ensure that the image is transferred correctly onto the sheet; i.e., the time between two sheets at the print head must be sufficient, and no sheets should collide with each other at the merge point. A timing designer needs to develop, check, and optimize the sheet timing as well as the timing constraints so that the displacement of sheets is sound and complete. In addition, the productivity KPI is influenced directly by the design of the sheet timing and sheet constraints.

The demand for higher productivity has led to larger and more complex paper paths, as mentioned at the start of this section. The numerous timing requirements that arise due to this additional complexity are harder to manage than when *HappyFlow* was originally developed [12]. Whereas previous printers had up to 10 simultaneously moving sheets, paper paths of current printers may hold 100 moving sheets simultaneously. The variety of media types (such as sheet dimensions and material type) has also increased. The interactions between the sheets can therefore no longer be optimized or analyzed without advanced tool support.

The definition of these scheduling constraints was lifted from C++ constructs. Similar to in the sheet timing models, the KernelF expression language was embedded to allow sufficient expressiveness without having to create our own

```
Inter-sheet constraints
inter-sheet constraint HFDeflectorConstraint(prev: Sheet, next: Sheet) on Loop
  TE (prev) = OCMDEF1(1) ----- parameter(sec.switchTime) s ----> LE(next) <= OCMRPI1(1)
    CG ref POI: ITMSREULPI

Initial sheet constraints

initial-sheet constraint WarmUp(s: Sheet)
  LE(s) <- ITMIN(1) for at least: max(parameter(sec.DrumWarmUp), parameter(sec.HeadWarmUp)) s

Distance constraints
[ Code gen: DISABLED ]
distance constraint Test(prev: Sheet, next: Sheet) on segment Input
  TE(prev) - LE(next) >= 0 mm
```

Fig. 9 Example sheet constraints in MPS showing how the minimum time/distance constraints are computed and how these relate to the sheets as well as the layout of the printer

implementation of all arithmetic, corresponding interpreters, and type checkers. This has made it possible to write down timing requirements succinctly, as shown in Fig. 9. Each timing constraint is generated into dozens of lines of C++ code. These sheet scheduling constraints are the input for heuristic online schedule optimization algorithms [18, 19, 21].

We have also added algorithms to perform automatic verification of sheet movement schedules. Lifting the models from C++ to MPS has also allowed us to implement an automatic verification technique. This verification technique can determine the *robustness* [6, 7] of a schedule of sheets with respect to the sheet scheduling constraints that were defined. The results of the verification can be shown directly in the model editor, due to the inline integration of graphs.

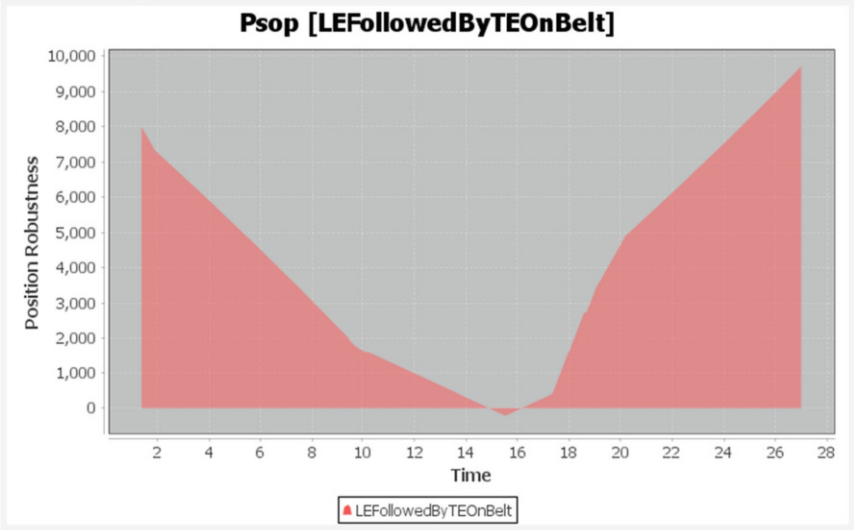
The position time diagrams resulting from the models in Sect. 3.3 can be interpreted as STL[4] signals for which we can write logic to check them for functional correctness, such as shown in Fig. 10. In this example constraint, the trailing edge of sheet 2 violates the constraint that it needs to arrive on a belt 0.2 s after the leading edge arrives on that same belt. The robustness is negative between $t = 15$ and $t = 16$, indicating that the constraint is violated at those time instances.

3.5 *Product Variants and Modularity*

Section 2 discussed the increasing variation in printer configurations. We have already illustrated that the models for a particular configuration benefit the printer designers. However, due to the large number of possible configurations, we have also used modularity in the design to avoid repetitions. The modules imported from

[4]The extension of Signal Temporal Logic (STL) [6, 7] for constraints validation is developed by the Embedded Systems Institute (ESI) at the Netherlands Organisation for Applied Scientific Research (TNO) in the Octo+ research program with Canon Production Printing as carrying industrial partner.

```
plot LEFollowedByTEOnBelt
```




```
STL[ LEafterStartBelt ]: sheet2 LE >= 7200
STL[ LEbeforeEndBelt ]: sheet2 LE >= 8500
STL[ TEafterStartBelt ]: sheet2 TE >= 7200
STL[ TEbeforeEndBelt ]: sheet2 TE >= 8500
STL[ LEOnBelt ]: LEafterStartBelt && LEbeforeEndBelt
STL[ TEOnBelt ]: TEafterStartBelt && TEbeforeEndBelt

STL[ TEOnBeltWithinInterval ]: G( TEOnBelt ) between [ 0.2 , 5.2 ]

STL[ LEFollowedByTEOnBelt ]: LEOnBelt => TEOnBeltWithinInterval
```

Fig. 10 Example sheet constraint robustness using sheet displacement as STL signal

mechanical CAD drawings are enriched with other models regarding parts, timing, routes, etc. The paper paths and parts are composed from these enriched modules, in accordance with the information in the variability model, as shown in Fig. 5. This has led to a cleaner and smaller definition of the product lines than is possible without the variability models.

3.6 Continuous Integration of Model-Based Design Artifacts

The MSI language set described in the previous sections is used for generating packages of embedded software, simulation, and visualization artifacts. We have integrated the MPS command line build tool in our DevOps environment to achieve Continuous Integration. This is done in a three-stage approach:

1. The language engineers create and update the languages, which are built to the (MSI) DSL plugins.

2. The domain experts use these DSL plugins to create and update (MSI) models, which are generated into artifacts such as (C++) source code and XML files.
3. The generated artifacts are integrated into the embedded software repository to compile the final implementation.

The generated software packages therefore become immediately available to the embedded software developers. The versions are stored as build artifacts that can be referenced by multiple projects. The automated testing of languages (as part of step 1) and models (as part of steps 2 and 3) has ensured that multiple developers can work on the same code base (both models and languages) with the same conveniences as Continuous Integration allows for normal software development.

3.7 Concluding Remarks on MSI

We are using the projectional editor of MPS to combine textual, graphical, and tabular notations into a single editor experience. Combined with the modularity of MPS, i.e, the composability of languages that is possible due to the projectional nature, languages can quickly adopt existing solutions, such as mathematical notations for expressions, and inline integration of graphical editors and computed graphs.

4 Hardware-Software Interface

The Hardware-Software Interface (HSI) is a collection of DSLs for describing the available input and output (I/O) ports in the embedded system (electronics) boards of a printer. Such boards include, among others, an embedded processor and Field Programmable Gate Arrays (FPGA) that are to be programmed with embedded and control software. The top part of Fig. 11 illustrates that the HSI combines the

- Hardware domain to cover the I/O available in the embedded processor
- Hardware Description Language (HDL) domain for the I/O on the FPGAs
- Software domain for configuring the I/O

into a single domain-specific model. Each domain expert uses the respective projection to view and edit the part of the model that matches his/her domain. This allows the exclusion of the details of other domains, which are typically irrelevant for him/her.

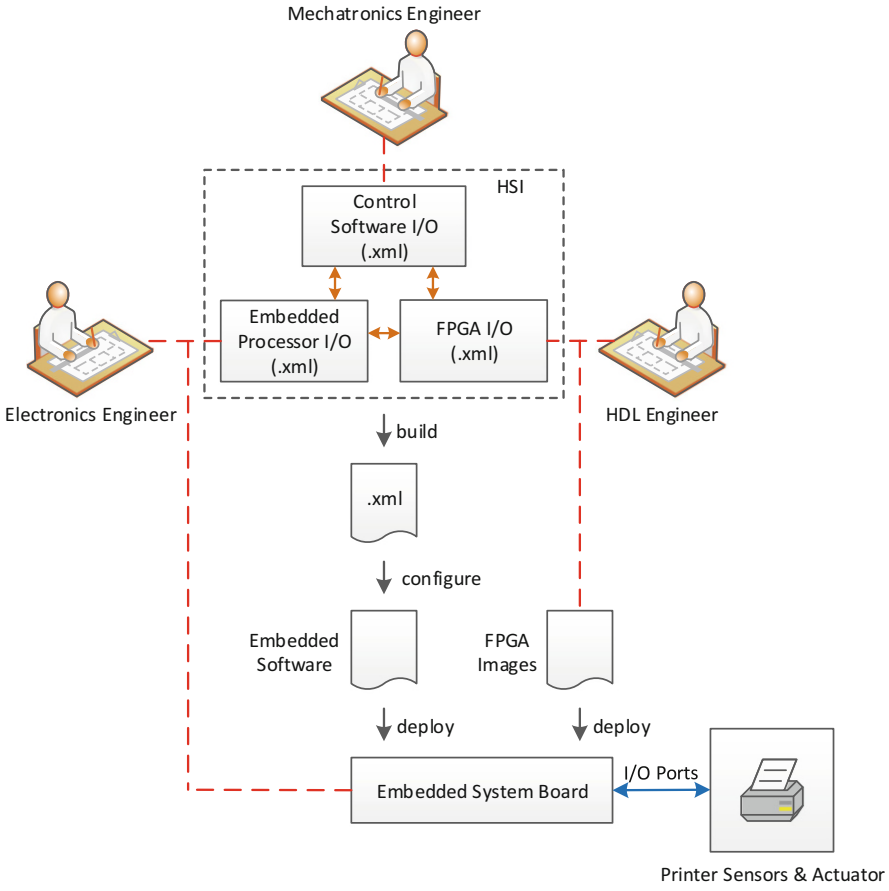


Fig. 11 Initial XML-based version of the HSI bridged three domains at design-time

4.1 Initial Version of HSI

The HSI was initially developed as DSLs described in XML files (see also Fig. 11). It relied on XSD and XSLT for syntax and sanity checking. Engineers in each of the domains would create their own set of files and the information in these files was combined *during build time* into one configuration file to be used on the printer. Three reasons were key to starting the approach with DSLs in MPS:

- **Model Size** Our projects use more and more I/O, which leads to an increase in model size and model connections. It became error-prone and too complex to keep the XML files consistent; the unique keys used to combine the specified information may not contain typos. Such typos were only detected at build time.

- **Modularization** The I/O is implemented by FPGAs and corresponding electronics in modules. The embedded system boards were modularized to allow for reuse, which avoids duplication of I/O modules and with it the risk of incorrect modifications.
- **Product Variability** Due to the product variability described in Sect. 2, the I/O that actually needs to be instantiated is no longer statically known at design and build time of the printer. Instead, the printer configuration is determined dynamically at run-time. The actual I/O that is used is thus only known at run-time. This means that with the initial setup, the super-set of all I/O that can be instantiated for each configuration needed to be available.

The complexity resulting from these key reasons amounted to increased complexity for the engineers. More complex files as well as more complex checking rules were needed to ensure that files used on the target (i.e., the embedded systems board) were still valid. Note that errors that occur in the I/O at run-time are hard to debug and in the worst case even lead to broken mechanical hardware that is expensive to replace. More complex designs in the FPGA were also needed to incorporate all the I/O, which in turn led to the need for bigger FPGAs and thus a higher cost price. The initial approach was not scalable for product lines that are increasing in complexity, while striving for a price-competitive product for the customer. To address these challenges, we created a new version of the HSI in MPS (see Fig. 12).

4.2 Combined Domains and Domain-Specific Editing

MPS enabled the combining of all the domain-specific knowledge and information covered in several DSLs into a single source of information. This means that MPS guards the consistency of the model on the fly, while still allowing the engineers to edit the information with the same domain-specific editing experience as before, by making use of projections. While the whole system has become more powerful and more complex, the editing of the HSI information has been kept simple.

4.3 Harnessing Modularization

While the modularization of the embedded system board introduced more complexity to the HSI, it also enabled the exploitation of the generation power of MPS. The HSI specification now generates the actual HDL content that is used to create FPGA images. In the initial version of Fig. 11, these FPGA images were made by hand for each specific situation and the HSI was written manually. This approach was cumbersome and error-prone in view of guaranteeing the consistency between the FPGA image and the HSI. With the new version of MPS shown in

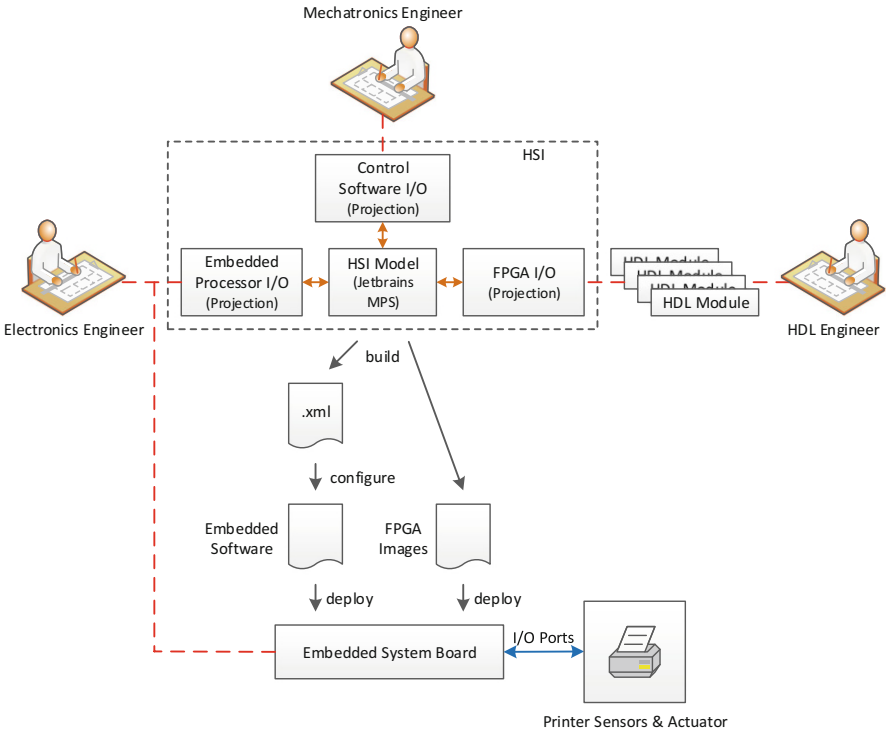


Fig. 12 MPS version of the HSI

Fig. 12, this process was reversed. The HSI now uses standard building blocks, each corresponding to an HDL module. The specification of such blocks leads to generating HDL code and thus to generating FPGA images that are consistent with the HSI.

Because MPS is also able to show the HSI in a simplified way, the new HSI version also enables non-HDL engineers to easily create an FPGA image just by specifying which I/O they need for their functionality or test setup. This automation frees up HDL engineers to work on novel design solutions instead of having to spend a lot of time on manually creating FPGA images for various printer and test setups.

4.4 Coping with Increased Complexity

The demand for modularity in our embedded systems board increased the complexity of the initial version of the HSI to a point where it made sense to migrate to a language workbench. The increase in complexity was tackled in multiple ways (see Fig. 12):

- **Projecting the same model in different ways** Different experts can now view and edit the same source of information without being bothered by the specifics of other domains.
- **Error reporting and quick-fixes/intentions** The initial version of the HSI had very limited error checking, and no way of helping an end user with solving detected errors. The HSI DSLs in MPS now make it possible for an inexperienced user to create an HSI instance, and if an error is introduced the editor can be used to help the user to resolve such errors.
- **First-class support for language testing** In the initial version of the HSI, the end-to-end functionality was checked by comparing the output for a set of predefined input files against a reference output. Such a check is still in place to validate the generators of the HSI DSLs in MPS, but now the editor experience and checking rules are also tested. Such a mature regression test allows new functionality or redesigning to be added without much risk on regression.

4.5 Status and Outlook for HSI

The development of the HSI has taken approximately one man year of effort over a period of 4 years by one domain expert/language engineer. During this period the HSI has been adopted by the projects within Canon Production Printing as functionality has become available. At this moment it is used in all main projects for the expansion and maintenance of the HSI and VHDL development. The total user base is approximately one dozen engineers, while the maintenance and new development is done by one language engineer.

The exploitation of MPS has automated ensuring the consistency across different engineering disciplines. Other information sources that include information about I/O needs is captured by other DSLs in MPS, such as the sensors and actuators in the Mechanics-Software Interface (see Sect. 3). Such information can be used to further facilitate the definition of an HSI instance. On the other hand, it would also be possible to use the HSI with all its information on the I/O as the input for diagnostics software to support service engineers. Finally, the information could be used to automatically create I/O tests for development purposes.

Mechatronics engineers at Canon Production Printing use Matlab Simulink for model-based development of the control software to be run on the embedded system boards. They use the available simulation facilities of Matlab Simulink to validate the model against possible inputs. In the past, these models were manually rewritten into C or VHDL (depending on the deployment). This meant that the validation had to be redone to ensure that the C or VHDL code was correct. Nowadays, automatic generators are used when targeting C code. It is also possible to generate VHDL code from the Matlab Simulink models. However, such generated VHDL code does not integrate straightforwardly with the VHDL code already generated by our MPS-based version of the HSI. Hence, some standardization on the interface with Matlab Simulink models to generate the required glue VHDL code is being investigated.

5 Domain-Specific State Machine Specification

As in many high-tech companies, control behavior ("state machines") is an important aspect of the software that we develop at Canon Production Printing. In our software development process, many different approaches have been and are being used to implement such behavior. This ranges from switch-case statements in general-purpose languages, standard state machine libraries, in-house built state machine libraries, in-house built state machine DSLs and tools (implemented using either generic scripting languages or language workbenches), to the use of advanced commercial tools such as RSARTE [13]. Having such a multitude of different approaches to tackle the same challenge is not desirable. We therefore aim toward preferably one standard state machine approach and tool.

5.1 Yet Another State Machine Language

The concept of state machine is "horizontally" domain specific in the sense that it is not specific to a particular "vertical" business domain like printing. One would expect to be able to buy a perfectly suitable off-the-shelf tool. We have, however, not been able to find a tool that fits all our needs in the different contexts where control software plays a role. We therefore took up the challenge to create a DSL that does suit our needs. Together with several of our experts in control software development, we thought about how we would want to specify some of our most complex control components. In parallel, we developed a DSL that enabled us to write down such specifications.

5.2 Open Interaction Language (OIL)

An important requirement on the DSL was that it should enable us to compactly and intuitively express the complex control behavior that we face. A crucial factor was the ability to separate the behavior into multiple concerns, each describing a specific aspect of the behavior. Furthermore, next to a compact and intuitive textual representation, it should also allow for compact and intuitive graphical representations, all of which should be semantically consistent and complete. The language should have a well-defined formal semantics and allow for behavioral verification and fully automatic generation of efficient executable code. The resulting language was called OIL, which stands for Open Interaction Language.

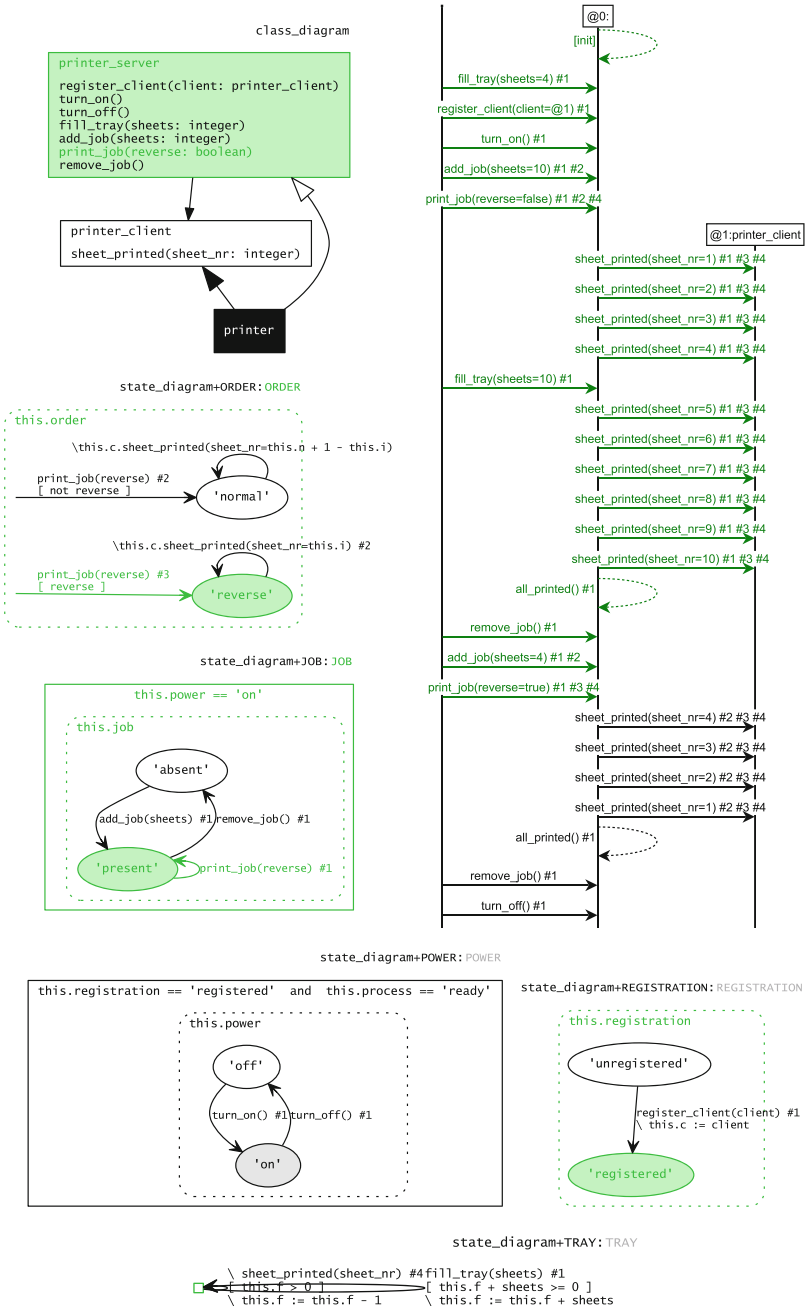


Fig. 13 Some views of a toy OIL specification in the Python/XML-based prototype

5.3 Prototype Tool

To develop and use OIL, we created a prototype tool that can generate various views of a specification such as class diagrams, state diagrams, sequence diagrams, implementation code, and model-checking code. The prototype was implemented with generic technologies such as Python and XML/HTML. An intricate interplay of parsers and code generators enables aspect-based views and simulation of traces (Fig. 13).

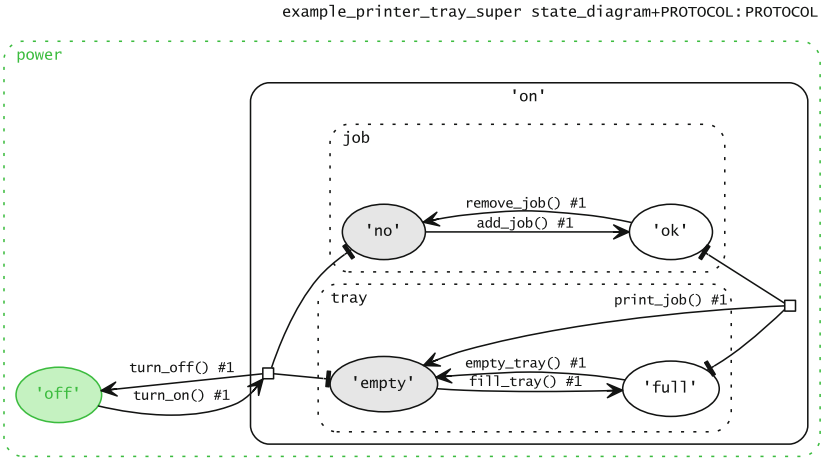
5.4 Behavioral Verification

To enable behavioral verification of OIL specifications, a PhD project was started in which a transformation from OIL to the formal language mCRL2 was developed [3, 9]. Using the mCRL2 model-checking tools, several behavioral aspects such as absence of deadlock, liveness, and confluence can be automatically verified. Using simulation, counterexamples can be analyzed at the level of the specification.

5.5 Usability and Maintainability

Although the prototype tool for OIL has quite advanced capabilities, it is just a prototype, and its usability and maintainability are mediocre. To tackle this issue, but also to extend our knowledge on domain-specific modeling and application of formal methods in general, two PhD students and a masters student have together studied OIL in depth, one focusing on syntax, one on semantics, and one on code generation. The language workbench Spoofax was used to create a more maintainable implementation of OIL, including generators to several target languages [4, 5, 10].

Although Spoofax is a good tool for implementing textual DSLs and transformations between DSLs, many of our engineers prefer a tool that enables them to directly define and manipulate state machines in their graphical form. This is where MPS came in. Based on the OIL language, an MPS implementation including textual and graphical projections was constructed. Thus, instead of having a textual DSL from which graphical representations were generated, we now have an integrated tool in which both textual and graphical projections can be directly edited (Fig. 14).



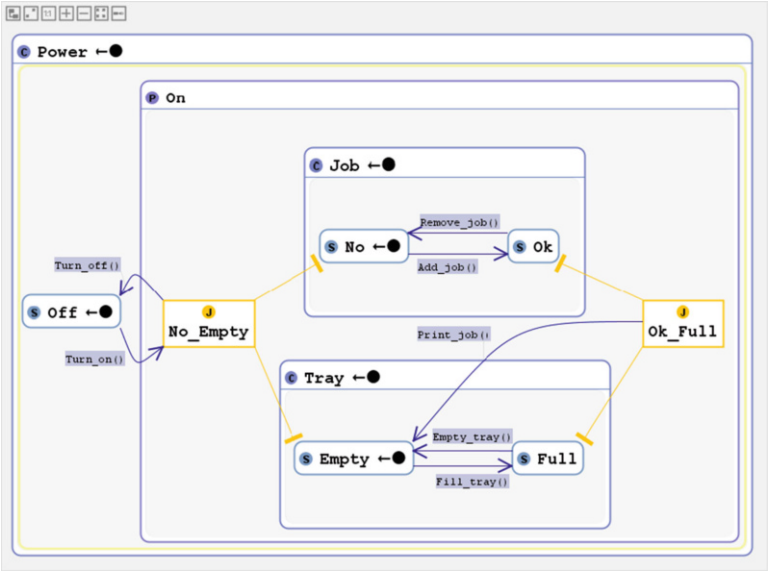


Fig. 14 An OIL specification in the Python/XML-based tool (top) and MPS-based tool (bottom)

5.6 Conclusion for OIL

There is still quite some work to do. The current MPS implementation only supports a subset of OIL, lacking several features that are essential to keep specifications of complex behavior compact. The reason for this is that the currently implemented code generator in MPS generates code that uses an in-house developed C++ state machine library. This library lacks several features of OIL and is also approximately one factor 10 slower than the low-level C++ code that is generated by the prototype.

We also implemented an OIL specification text generator in MPS, enabling use of the prototype tooling and code generator. However, being a prototype, this code generator is not something we want to maintain. The work carried out by the PhD students and master's students resulted in a flexible code generator architecture. This architecture uses several intermediate representations to ease adding code generators for different target programming languages such as C++, C#, and Java. As mentioned, this was implemented in the Spoofax language workbench. It remains an open question how easy it is to implement this in MPS, or whether we perhaps should resort to a combination of MPS as editing front-end and Spoofax as code generator back-end.

Overall, MPS enabled us to create a state machine tool that our engineers value for its capabilities. Especially the idea of not having separate code and design, but code *generated from* the design, combined with the ability to model graphically was well-received. Full integration of the tool into our development environment was also an important factor for its acceptance, although properly integrating it took quite some effort. There are still open challenges, like a proper approach to code generation and integration of behavioral verification. Most of the conceptual challenges were, however, already tackled in the prototype, and thus creating our one standard state machine approach and tool should be a matter of (hard...) work.

6 Virtual Printer Configuration

Canon Production Printing is creating various virtual printer environments to support the development of novel printer systems. The Virtual Printer Configuration DSLs define simulation specifications that allow the evaluation of trade-offs in aspects such as *productivity*, *print quality*, and *cost* of a printer. We describe how MPS helps us to manage simulation configurations and show two examples of simulation specification DSLs specific for our virtual printer environment that supports development of novel print processes: *Print Head* and *Carriage Motion* specifications.

6.1 Specifying Simulation Configurations

Figure 15 highlights our virtual printer environment that supports the development of novel print processes. It processes a *source image* through various simulation models that represent the steps in the print process such as *image processing*, *positioning* of the print heads and medium (e.g., paper), and *jetting* ink onto the medium. After ink droplets have been jetted, they (potentially) *interact* with other droplets and the medium. *Solidification* of the ink (i.e., the ink turns from a liquid into a solid substance by applying, for example, heat or UV light) is the last step in creating a *virtual print*. Subsequently, relevant properties of the virtual print can be evaluated. This includes *image quality aspects* (e.g., how good the printed image looks), *print robustness* (e.g., how well the ink sticks to the medium), and *media deformation* (e.g., how well the printed medium keeps its shape after being wetted by the ink and dried by heat). *Design-space exploration* can be applied to find suitable parameter settings for the desired balance between the mentioned properties of the virtual print.

The Virtual Printer in Fig. 15 exploits several simulators and models, which each has a value on its own. Together, they allow for evaluation of a complete print process for different types of printer systems. The Virtual Printer Configuration DSLs allow the specification of what printer system type is being investigated and what further parameter settings to use during the simulations. This also means that different simulators or models are, for example, used for *single-pass* latex-inkjet printers such as the Canon VarioPrint iX product family,[5] which print sheets of paper in a single pass, and *multi-pass* UVGel-inkjet printers such as the Canon Colorado product family,[6] which use a similar multi-pass print principle to commodity desktop printers. A concrete configuration expressed in the Virtual Printer Configuration DSLs results in instantiating the relevant chain of simulators with the desired parameter settings. Instantiated simulator chains are typically a variation of the following example:

- Select/generate input image.
- Process the input image for a selected print strategy.
- Compute media/print head movement trajectory to position ink droplets.
- Simulate nozzle activation, ink spreading, and solidification.
- Evaluate image quality aspects.

We aim to make the simulation models usable for both experts and novices. This can be achieved by mixing high-level and low-level detail in the configuration of each of the simulators. For novices, we use MPS to show only those abstractions that are required from the printer design point of view. The remaining parameters are then auto-generated or have default values. Experts can use either their familiar

[5] https://www.canon-europe.com/business-printers-and-faxes/varioprint-ix-series/.

[6] https://www.canon-europe.com/business-printers-and-faxes/colorado-1640/.

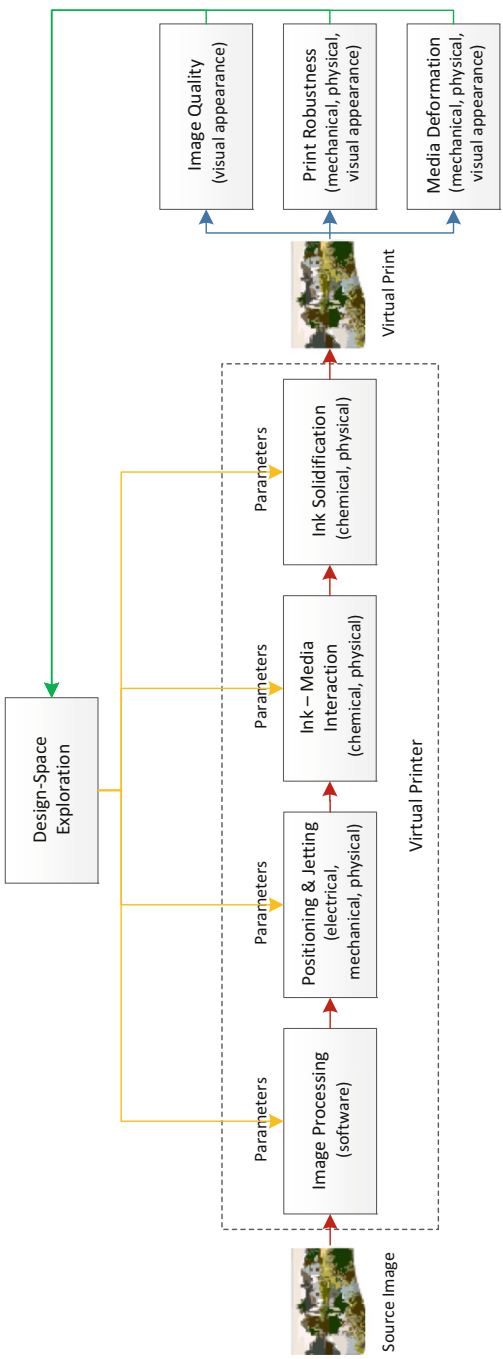


Fig. 15 Virtual printer supporting evaluation of print quality aspects during print process design

interface to the simulation tool or model directly, or use the more detailed configuration facilities provided with the Virtual Printer Configuration DSLs. MPS allows us to present the configuration models via table/text-based editors and a graphical editor using a similar view to that shown in Fig. 15, showing the various simulation models (with ability to edit their parameters) and their interconnections.

6.2 Example Virtual Printer Configuration DSLs

Several parts of a printer need to be specified to allow for simulation of its print quality and productivity. The placement and activation of *nozzles* in a *print head* have a significant impact on where the ink droplets can be jetted, and on the resolution and accuracy of droplet positioning. This also largely determines the productivity or throughput in terms of, for example, printed sheets or area per time unit. Given a model of the ink spreading behavior and the behavior of interaction with the medium, the ink droplets jetted from print head nozzles simulate how ink layers are formed on the medium [22, 23]. This is essential for evaluating the print quality of the final print (i.e., its image quality, print robustness, and media deformation; see Fig. 15).

6.2.1 Print Head Specifications

The first example of our Virtual Printer Configuration DSLs concerns the specification of print heads. Such *print head specifications* include the following aspects:

- Ideal (designed) position of nozzles in print heads
- Mapping of image lines onto the nozzles to activate
- Imperfections in nozzle positions and jetting conditions

Due to the large number of nozzles inside a print head, it is not desired to specify each individual nozzle separately. In addition, specifying each nozzle separately would hide the overall structure of nozzles in a print head. Hence, we use MPS to define logical structures of nozzles and give immediate feedback to the user in terms of correctness and derived properties of the nozzle structure definitions. We also interactively visualize the effective placements of nozzles, so that the end user can verify the interpretation of these logical structures as illustrated in Fig. 16.

After defining the ideal nozzle locations for all print heads, the user can specify how the image lines are mapped onto nozzles. These mappings rely on a synchronization between the Image Processing step and the nozzle activation simulator in the Positioning & Jetting component shown in Fig. 15. The third aspect of the Print Head Specification model, i.e., nozzle position and jetting imperfections, is defined on top of the ideal nozzle locations. Such imperfections can be *structural* or *dynamical*. Structural imperfections include misplacement of nozzles, and scaling

```
virtual printer configuration Examples using Scanning coordinate system
{
  staggered nozzle plate ExampleChip {
    10 nozzles at 75 npi in  x
    Stagger width: 1 mm
  }
  nozzle group ExampleHead {
    placed ExampleChip at (x, y) = (0, 0)
  }
  Print Carriage 4xExampleHead @ 32 kHz {
    2 drop sizes

    Ink Channels: C M Y K
    Slot 1 : ExampleHead at (x, y) = (0, 0) using C
    Slot 2 : ExampleHead at (x, y) = (4, 0) using K
    Slot 3 : ExampleHead at (x, y) = (8, 0) using M
    Slot 4 : ExampleHead at (x, y) = (12, 0) using Y
  }
  Print Carriage 8xExampleHead @ 21 kHz {
    2 drop sizes

    Ink Channels: C M Y K
    Slot 1 : ExampleHead at (x, y) = (00, 0.000000000) using Y
    Slot 2 : ExampleHead at (x, y) = (02, 0.169333333) using C
    Slot 3 : ExampleHead at (x, y) = (04, 0.000000000) using M
    Slot 4 : ExampleHead at (x, y) = (06, 0.169333333) using K
    Slot 5 : ExampleHead at (x, y) = (08, 0.000000000) using K
    Slot 6 : ExampleHead at (x, y) = (10, 0.169333333) using M
    Slot 7 : ExampleHead at (x, y) = (12, 0.000000000) using C
    Slot 8 : ExampleHead at (x, y) = (14, 0.169333333) using Y
  }
}
```

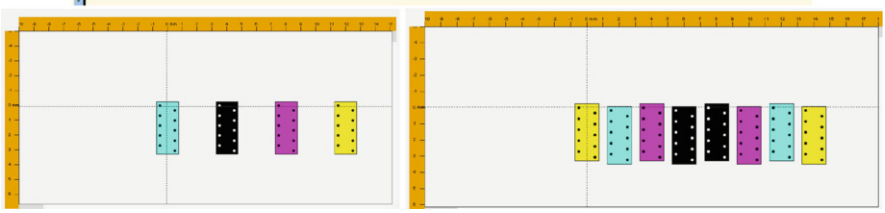


Fig. 16 Example printer configuration and visualization of 4 × ExampleHead and 8 × ExampleHead in MPS

or rotation of nozzle structures. Dynamical imperfections include *droplet volume deviations*, definition of so-called *satellite droplets*, and *jetting speed variations*.

Separating the specification of the ideal (designed) print head properties from possible imperfections enables the configuration of the simulator models in a way that would not be straightforward when configuring the simulators directly. This is because imperfections need to be combined with the ideal (designed) situation before configuring the simulators. For example, some imperfections may end up as an addition or multiplication for a single parameter of a simulator. Furthermore, MPS helps us in guiding the users in understanding constraints when configuring the simulators. Certain imperfections can, for example, not occur together (i.e., they conflict with each other), and that information is presented to the user while editing.

6.2.2 Carriage Motion Specification

The nozzles defined in the Print Head are either placed in a *Print Carriage* in case of multi-pass printer systems or in a *Print Station* in case of single-pass printer systems. Multi-pass print systems move their carriage(s) over the same piece of media multiple times to create the result image. This approach requires fewer of the costly print heads, and allows for much larger and a much wider range of media to be printed, at the cost of additional time to print the resulting image (lower productivity).

To properly simulate the forming of ink droplets, the relative timing between droplets needs to be accurately modeled. Section 3.3 describes our approach to modeling how a piece of paper flows through single-pass cut-sheet printer systems. In such systems, the sheet moves at a constant velocity when the image is being printed. This is actually a very small part of the overall sheet movement from the *paper input modules* to the (printed) *paper output modules* of such printer systems.

For multi-pass printer systems, the carriage motion needs to be described. The motion of the carriage(s) is not necessarily with constant velocity and has a significant impact on the relative timing between droplets. We describe the synchronization points of the carriage(s) together with the minimum and maximum physical constraints (acceleration, velocity, time) imposed on each of the trajectories. The timing and trajectories of each cycle are automatically computed by the simulator. MPS again allows us to mix convenient notations for the synchronization points, parameters, and visualization of the carriage motion trajectory (Fig. 17).

6.3 Outlook for Virtual Printer Configuration

At the time of writing, we are progressing with development of the simulation infrastructure, as well as creating several domain-specific simulators and DSLs to configure these simulators. Only recently, two language engineers started developing the DSLs described in this section. Hence, they are still in an early prototyping phase. We are, however, already deploying these DSLs together with the simulation infrastructure in production contexts. Currently, we are broadening the scope of users from mostly software developers, model creators, and simulator experts to application specialists. At the time of writing, there are about five application specialists that exploit the Virtual Printer in Fig. 15 in daily printer development activities.

Looking at maturing the DSLs described in this section, variability modeling would allow us to specify which combinations of simulation models are allowed, and give the user a convenient interface to define a virtual printer configuration for their experiment. This will allow for the easy interchanging of, for example, the print heads and the inks, the lamps in the curing carriage, as well as certain image processing steps. Hence, we intend to exploit feature models as described in Sect. 2.

```
timing notebook ExampleMotion (media width: (25.4 * 5) mm) {
  val headHeight = 127 / 225 * 25.4
  val headWidth = 10
  val headSpacing = 12

  val mininmalTurnTime = 0.700

  carriage geometry PrintingCarriage : # AEAEAE {
    PrintHeads(x: 0, y: 0, w: 4 * headSpacing, h: headHeight) at 0 , 0
    Y(x: 0 * headSpacing, y: 0, w: headWidth, h: headHeight) relative to top left of PrintHeads
    C(x: 1 * headSpacing, y: 0, w: headWidth, h: headHeight) relative to top left of PrintHeads
    M(x: 2 * headSpacing, y: 0, w: headWidth, h: headHeight) relative to top left of PrintHeads
    K(x: 3 * headSpacing, y: 0, w: headWidth, h: headHeight) relative to top left of PrintHeads
    PinCureR(x: 15, y: 0, w: 10, h: headWidth) relative to top right of K
  }
  carriage geometry CuringCarriage : # FFFFFF {
    PostCure(x: 0, y: 0, w: 100, h: 50) at 0 , 0
  }

  repeated motion SomeMotion {
    carriages: PrintingCarriage
    situations:
      BoS L2R {
        PrintingCarriage { right of PrintHeads is aligned with left of PrintMedium @ 1000 mm/s }
        CuringCarriage   { left of PostCure is aligned with left of PrintHeads     @ 1000 mm/s }
      }
      EoS L2R {
        PrintingCarriage { left of PrintHeads is aligned with right of PrintMedium @ 1000 mm/s }
      }
      BoS R2L {
        PrintingCarriage { left of PrintHeads is aligned with right of PrintMedium @ -1000 mm/s }
      }
      EoS R2L {
        PrintingCarriage { right of PrintHeads is aligned with left of PrintMedium @ -1000 mm/s }
      }
    trajectories:
      constant speed for PrintingCarriage : BoS L2R -> EoS L2R
      constant speed for PrintingCarriage : BoS R2L -> EoS R2L
      turn trajectory for PrintingCarriage : EoS L2R -> BoS R2L
        /max accel: 10000, max jerk: <no maxJerk>/
      turn trajectory for PrintingCarriage : EoS R2L -> BoS L2R
        /max accel: <no maxAccel>, max jerk: 1000/
    timing constraints:
      EoS L2R -> BoS R2L { time >= mininmalTurnTime }
      EoS R2L -> BoS L2R { time >= mininmalTurnTime }
  }
}
```

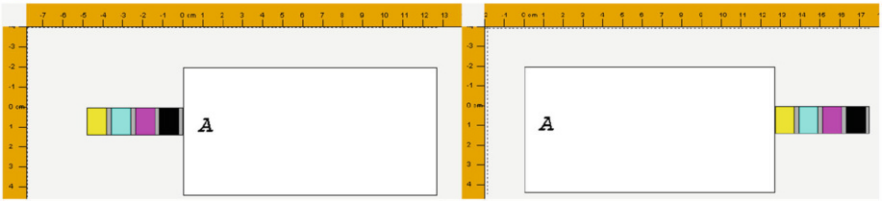


Fig. 17 Example carriage motion definition and corresponding situation visualization in MPS

We make use of the different projection capabilities of MPS, as well as the typical language engineering aspects such as error handling and type checking. We do not expect typical (non-software developer) users, i.e., application specialists exploiting the Virtual Printer in Fig. 15, to become comfortable with the user interface of MPS. Even with a stripped down RCP application, the user interface remains too focused on being a software IDE and language workbench. Another concern is model management. Typical application specialists may have some Matlab or Python programming experience, but most are not familiar with Git or other software version control systems. Avoiding or resolving version conflicts early is, however, an important aspect of model management. An approach to both avoid the use of complicated version control systems and improve on user interface aspects is the Modelix solution direction for collaborative modeling described in the next section.

7 Collaborative Domain-Specific Modeling

MPS comes with a high-end Integrated Development Environment (IDE) that is well suited for the development and exploitation of Domain-Specific Languages (DSL) in an industrial context. This IDE is a stand-alone desktop application. Its users include *DSL engineers* developing DSL models with accompanying functionalities and *DSL users* that create and exploit instances of such DSL models. The appearance of the IDE can be customized to ease usage for DSL users (e.g., as RCP). This helps in reducing the learning curve that is often experienced and in only covering the specific functionality relevant for DSL users. Exploitation of MPS at Canon Production Printing has, however, revealed the need for some additional capabilities:

- **Collaborative modeling**: DSLs at Canon Production Printing often represent domain-specific interfaces between models from different engineering disciplines, as also exemplified with the DSLs in Sects. 3 and 4. Such interfaces would benefit from the ability to collaboratively update DSL instances in a similar fashion to Google Docs and Microsoft Office 365 support for office documents. Collaborative modeling would benefit from a web-based front-end for, in particular, DSL users. A server-based deployment would also ease version control for DSL users unfamiliar with traditional software technologies and terminology for version control. In addition, it eases updating any involved tools (e.g., for automated code generation, build, and test tool chains) in case of DSL evolution.
- **Integration in larger GUI applications**: Canon Production Printing has various existing development environments that could benefit from exploiting DSL technology. Such environments may rely on custom Graphical User Interfaces (GUI) for which it is often not easy or even infeasible to (completely or partly) replace them in a gradual or disruptive step by the IDE for MPS. Such situations

would benefit from the ability to integrate the DSL technologies provided by MPS into the existing development environments as DSL widgets.

- **Domain-specific customization of look-and-feel**: Canon Production Printing positioned model-driven development as crucial for timely development of novel printer systems. Engineers of all engineering disciplines are expected to exploit DSLs for which tooling is realized using MPS. However, engineers with little or no affinity for software development have many difficulties in adopting the IDE (even as DSL users). It would be beneficial to provide DSL technology as part of highly customized GUIs with a domain-specific look-and-feel that is much closer to that of domain-specific tools currently used by such engineers.

This section[7] peeks into the future direction Canon Production Printing is considering for a widespread exploitation of MPS throughout the organization, which covers not only research & development but also, for example, sales and service.

7.1 Blended Collaborative Domain-Specific Modeling

A fundamental concept underlying MPS is the use of a *model-view architecture*. This is exposed by the DSL model instance being *projected*, possibly even with multiple different syntaxes at the same time, to a DSL user. The DSL user can change the DSL model instance via any of these projections or *views* in traditional model-view architecture terminology. This results in the DSL model instance changing, and hence all projections of the DSL model instance update consistently. This principle is also suited for collaborative modeling since it is irrelevant whether the projections are to a single DSL user or to multiple DSL users. Such projections may concern different syntaxes for the same DSL model instance. Note that *blended collaborative domain-specific modeling* as described here requires the one DSL model instance to be (simultaneously) accessible for all involved DSL users.

itemis is experimenting with generating all that is needed to support blended collaborative domain-specific modeling based on a DSL model defined in the IDE that comes with MPS [1]. Figure 18 shows a high-level overview of *itemis*' initiative called *Modelix* [15]. It relies on the DSL model instance being accessible on a central server. Based on a plugin, the DSL model instance can be accessed with the existing IDE of MPS. The DSL model instance can, at the same time, also be accessed via a web-based front-end in a DSL user's web browser. The idea is that execution of DSL facilities such as model transformations and code generation runs on the server.

itemis' Modelix approach shown in Fig. 18 has the major benefit that existing DSL models defined with MPS like those highlighted in previous sections can

[7]The work described in this section is partially supported by ITEA3 project 18006 BUMBLE.

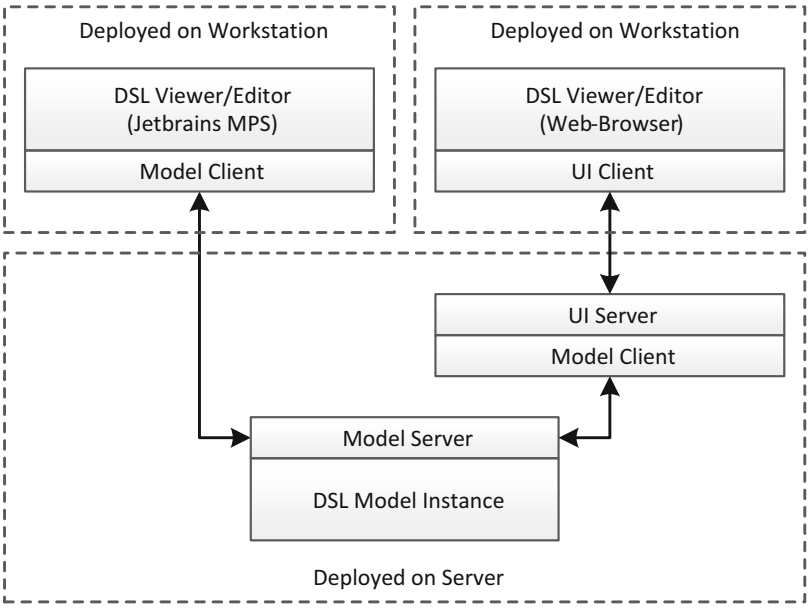


Fig. 18 A model-view architecture for blended collaborative domain-specific modeling

be used in a blended collaborative way without restarting their development in a different technology such as Javascript. In addition, the existing IDE can still be used by DSL users that are already familiar with it as it integrates seamlessly. Based on these advantages, Canon Production Printing intends to use *itemis*' technology to extend the applicability of DSL technology throughout a larger part of the organization.

7.2 DSL Widgets in Custom GUIs

Next to the Graphical User Interfaces (GUIs) made available to print professionals (customers of Canon Production Printing) as part of the Professional Digital Printer product families, several tools to develop and maintain these product families are also being created for use within Canon Production Printing. Application of DSL technology to formalize the domain-specific knowledge underlying both such Graphical User Interfaces has major benefits. However, existing DSL technology does not (yet) provide the customization flexibility that traditional software technology provides to develop GUIs. With the introduction of *itemis*' Modelix, Canon Production Printing envisions the use of Modelix to create DSL widgets as part of traditional GUIs applications realized with traditional web technology such as Google's Angular [11].

MPS and Angular use a similar component-based approach to compose complex views on a collection of related data items from simple views on individual data items. On the other hand, the concepts underlying a model-view architecture can be realized fairly easily in Angular. In the Angular context, the controller concept of a model-view architecture, which is responsible for converting (raw) data into a form that can be displayed to users via a view, is often denoted as *ViewModel*. Given the similarities, Canon Production Printing envisions that Angular applications can be partly DSLified with Modelix-based components, bringing together the strengths of MPS's DSL technology and the customization flexibility of Angular.

7.3 Outlook for Collaborative Modeling

Canon Production Printing intends to investigate how the combination of Modelix and Angular can serve the creation of blended collaborative domain-specific modeling as part of (possibly existing) larger GUI applications with domain-specific customization and look-and-feel. This future direction is expected to vastly increase adoption of MPS's DSL technology at Canon Production Printing.

8 Conclusions and Lessons Learned

Engineers can spend countless hours discussing a design aspect of a printer system, both within and across engineering disciplines. It is hard to overestimate the time spent on discussing printer-specific details while (sometimes unconsciously) speaking different languages to explain such details to each other. Commodity tools used for capturing printer-specific knowledge in models do not necessarily allow for domain-specific customization. Hence, when sharing knowledge, engineers still spend a lot of time on interpreting the models in such tools. At Canon Production Printing, technology for domain-specific languages has proven to provide a suitable means to bridge this domain-specific interpretation gap between models in commodity tools.

Moving to a model-based way-of-working is mostly a no-brainer at Canon Production Printing. However, choosing MPS as core technology to bridge domain-specific interpretation gaps certainly is not. It is also not straightforward to introduce it as core technology for formalizing printer knowledge to enable automated processing. As highlighted in previous sections, the main challenges with MPS have been as follows:

- *Steep learning curve* of MPS, both for DSL engineers and DSL users: The projectional editing experience is unlike any commodity tool, which therefore requires adapting to a different way of user interaction. In addition, the vast amount of already available languages and their interrelations is challenging to

overview. These foundations are, however, key to the strengths of MPS. They enable language modularity and seamless editing of different views with one and the same underlying model. We believe that the direction of *itemis*' Modelix can substantially reduce the steepness of the learning curve for DSL users.

- *Lack of full-fledged DSL models in MPS for commodity languages* such as C++, C# and VHDL: As exemplified in Sects. 4 and 5, such languages are generation targets from specification models at Canon Production Printing. Instead of being able to exploit model-to-model transformations to full-fledged DSL models that rely on fully engineered model-to-text transformations in a final code generation step, we resorted to creating our own model-to-text transformations. This approach hampers maintainability, also in view of remaining compatibility with libraries for such target languages. Hence, we intend to exploit future DSL model solutions that the (open-source) MPS community may develop.
- *Existing parsers or grammar rules* commonly available in alternative technologies *are not immediately reusable* in MPS. Integrating existing textual languages, for example, tends to require more effort than just integrating an existing parser technology. We believe that extending MPS with a means to also allow traditional text-based parsing would ease adoption considerably.
- We experienced that *the performance* of (chains of) (model-to-model) transformations *can be undesirably low*. This is, however, not only experienced for MPS but also for alternative DSL technologies, including Eclipse Xtext/Sirius.

Despite the above, more and more people within Canon Production Printing are taking up the challenge to learn MPS or will be empowered by user-friendly interfaces toward a consistent and coherent set of domain-specific models (e.g., by exploiting *itemis*' Modelix). The key advantage over other DSL technologies to pursue this direction are the architectural foundations of MPS easing supporting (1) multiple syntaxes (views) for a DSL model (including having one source of information) and (2) composability or modularity of DSL models (including the availability of meta-languages like KernelF). This enables DSL engineers to focus on the DSL model itself instead of on resolving low-level, often tool-related, challenges that tend to arise in other DSL technologies. DSL users benefit primarily from the ease of using multiple syntaxes and of course the well-known general benefits of using DSL technology (e.g., the early while-you-type validation and auto-completion facilities).

References

1. Birken, K.: MPS Applications in the Browser: Cloud MPS (2020). https://blogs.itemis.com/en/mps-applications-in-the-browser-cloud-mps
2. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. ACM Press/Addison-Wesley, New York (2000)
3. Bunte, O., Willemse, T.A.C., van Gool, L.C.M.: Formal Verification of OIL Component Specifications using mCRL2 (2020)

4. Delft University of Technology: Spoofax. https://www.metaborg.org
5. Denkers, J., van Gool, L., Visser, E.: Migrating custom DSL implementations to a language workbench (tool demo). In: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE), pp. 205–209 (2018)
6. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Formal Modeling and Analysis of Timed Systems, pp. 92–106. Springer, Berlin (2010)
7. Donzé, A., Ferrere, T., Maler, O.: Efficient robust monitoring for STL. In: International Conference on Computer Aided Verification, pp. 264–279. Springer, Berlin (2013)
8. DSLFoundry: PlainTextGen. https://jetbrains.github.io/MPS-extensions/extensions/plaintext-gen/
9. Eindhoven University of Technology: mCRL2. https://www.mcrl2.org
10. Frenken, M.: Code Generation and Model-Based Testing in Context of OIL (2019)
11. Google: An Application Design Framework and Development Platform for Creating Efficient and Sophisticated Single-Page Web-Apps (2010–2020). https://angular.io
12. Heemels, W., Muller, G.: Boderc: Model-Based Design of High-Tech Systems; A Collaborative Research Project for Multi-Disciplinary Design Analysis of High-Tech Systems. Embedded Systems Institute (2006)
13. IBM: Rational Software Architect Real-Time Edition. https://www.ibm.com
14. itemis et al.: IETS3. https://github.com/IETS3
15. Lißon, S.: A Next Generation Language Workbench Native to the Web and Cloud (2020). https://github.com/modelix/modelix
16. Metzger, A., Pohl, K.: Software product line engineering and variability management: achievements and challenges. In: Future of Software Engineering Proceedings, pp. 70–84 (2014)
17. Pohl, K., Böckle, G., van Der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer Science & Business Media, Berlin (2005)
18. van der Tempel, R., van Pinxten, J., Geilen, M., Waqas, U.: A heuristic for variable re-entrant scheduling problems. No. 2 in ES reports. Technische Universiteit Eindhoven (2018)
19. van Pinxten, J., Waqas, U., Geilen, M., Basten, A., Somers, L.: Online Scheduling of 2-re-entrant flexible manufacturing systems. ACM Trans. Embed. Comput. Syst. **16**(5s) (2017). https://doi.org/10.1145/3126551
20. Völter, M.: Kernelf: An Embeddable and Extensible Functional Language (2017). http://voelter.de/data/pub/kernelf-reference.pdf
21. Waqas, U., Geilen, M., Kandelaars, J., Somers, L., Basten, T., Stuijk, S., Vestjens, P., Corporaal, H.: A re-entrant flowshop heuristic for online scheduling of the paper path in a large-scale printer. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE 15), 9–13 March 2015, Grenoble, France, pp. 573–578 (2015)
22. Wijshoff, H.: The dynamics of the piezo inkjet printhead operation. Phys. Rep. **491**(4–5), 77–177 (2010)
23. Wijshoff, H.: Drop dynamics in the inkjet printing process. Curr. Opin. Colloid Interface Sci. **36**, 20–27 (2018)