TNO innovation for life

Software Evolution of Product Families in High-tech Equip-ment

TNO Public › TNO 2025 R10901
26 August 2025

ICT, Strategy & Policy
www.tno.nl
+31 88 866 50 00
info@tno.nl

TNO 2025 R10901 – 26 August 2025

Software Evolution of Product Families in High-tech Equipment

Author(s)	Benny Akesson, Ben Pronk, Sven Weiss, David Worrell
Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	29 (excl. front and back cover)
Number of appendices	0
Sponsor	Hans Schurer
Project name	TechFlex
Project number	060.59531

All rights reserved
No part of this publication may be reproduced and/or published by print, photoprint, microfilm or any other means without the previous written consent of TNO.

Acknowledgement

The research is carried out as part of the TechFlex program under the responsibility of TNO-ESI in cooperation with Thales. The research activities are co-funded by Holland High Tech | TKI HSTM via the PPP Innovation Scheme (PPP-I) for public-private partnerships.

© 2025 TNO

Summary

Variability and evolution are key drivers of complexity in high-tech equipment. Every product has many variation points, resulting in a large number of unique system configurations. These systems must be maintained throughout their life-time, which may last several decades. During this time, digital technologies, e.g. containerization and orchestration technologies, may become deprecated or evolve many times. Updating configurations and implementations for each system variant to account for such changes is both costly and time-consuming. To make matters worse, changing software technologies affects technical system performance, requiring technical performance to be re-optimized and re-verified.

The TechFlex project is a collaboration between TNO-ESI and Thales that addresses the challenge of variability and evolution in software-intensive high-tech equipment. This is done by investigating to what extent it is possible to specify the software configuration in a technology-agnostic way yet automatically generate efficient software deployments that satisfy performance requirements. The result of this research is a model-based methodology to specification and automation with two steps.

1. Technology-agnostic specification of software configurations based on a family of domain-specific languages (DSLs) with technology-specific generators that produce artifacts to create a custom software deployment with minimum manual intervention.
2. Deployment optimization based on model-based reinforcement learning using Monte Carlo Tree Search (MCTS) that improves the mapping to software processes to compute nodes to ensure technical performance requirements are satisfied.

This report describes the methodology in context of a fictive Meal Delivery System (MDS), a simple case study inspired by an application in the defense domain. An evaluation of the approach shows that the technology-agnostic DSLs are expressive enough to describe the configuration of the MDS, as well as a confidential industrial case study.

Table of Contents

1 Introduction

The Netherlands has a vibrant high-tech equipment industry, active in many application domains. For example, consider ASML in Veldhoven, that produces lithography machines for manufacturing of computer chips, Philips Healthcare in Best, that produces medical equipment, such as X-ray and MRI machines, Canon that makes production printers in Venlo, and Thales making radars and fire control systems in Hengelo.

This industry is challenged by a number of trends that result in increasing system complexity of high-tech equipment across application domains. There is a trend towards growing *system diversity* in response to increasing market demands for customization, as every customer wants a system tailored to its specific needs [1]. Traditionally, systems have been made-to-measure for each customer, which makes it time consuming and costly to effectively develop and maintain systems throughout their life cycle. Industry has been addressing this challenge by moving towards a platform-based approach, where new customized products can be derived by configuring a general set of building blocks, much like Lego pieces are combined into unique creations. This change from engineering-to-order to configure-to-order is not just technical, but often comes paired with an organizational transition from a project organization to a product organization.

To meet the increasing demand for customization, systems are offered with many commercial options, resulting in a plethora of unique system configurations, also called *product variants*. Having a large number of variants is challenging for the industry due to the long life-times of high-tech equipment that often remains operational in the field for several decades [1]. During its operational life-time, the system is likely to outlive many of its constituent technologies, in particular digital technologies that evolve very quickly. For example, microservice architectures were not an established architectural approach fifteen years ago and commonly used containerization and orchestration technologies like Docker and Kubernetes were not around [2]. Note that systems that were built at that time are only about half-way through their expected operational life-time and who knows what technologies will be around in 15 years, when those systems are reaching their end of life?

The market expects modern systems to *continuously evolve* during their life-time to make sure they are kept secure, up-to-date, while continuously improving their functionality. Manually evolving a large number of product variants, e.g. to incorporate new and improved software technologies, requires substantial re-development effort and is very costly. After such updates, all variants must also be re-verified to ensure they still work correctly. Not only functional re-verification is required, but also in terms of non-functional requirements, such as system performance. This is challenging as performance aspects, such as timing, are emerging properties from interacting hardware and software components.

The TechFlex project addresses the challenge of reducing the time and cost associated with system diversity and evolution at the level of the software platform (product), with a focus on changes to programming languages, such as C++ or Java, or software deployment technologies, like Kubernetes and Docker. The project envisions a product-based approach with building blocks to quickly configure bespoke software solutions for each customer that always satisfy their (performance) requirements throughout their entire lifecycle. To achieve

this, the long-term goal envisioned by the project is a model-based methodology and supporting tooling that enables custom software configurations to be specified in a technology-agnostic manner, and from which a software deployment that satisfies (performance) requirements is automatically generated and regenerated, as software technologies evolve.

Towards this long-term goal, the TechFlex project investigates two research questions:

RQ1)To what extent can we generate efficient software deployments for different technologies from a generic software configuration model?

- To what extent can the generic software configuration model be decoupled from the deployment technology?
- Can it be completely decoupled while still generating efficient deployments, or only encapsulated in a separate technology-specific model?

1.

RQ2)How can we efficiently optimize software deployment(s) to satisfy performance requirements for a particular product configuration, load, and deployment technology?

- What are the benefits and drawbacks of performance prediction vs. performance evaluation?

The research has resulted in *a model-based methodology to specification and automation* with two steps, illustrated in Figure 1:

1. Technology-agnostic specification of software configurations based on a family of domain-specific languages with technology-specific generators that produce artifacts to create a custom software deployment with minimum manual intervention.
2. Deployment optimization based on model-based reinforcement learning using Monte Carlo Tree Search that improves the mapping to software processes to compute nodes to ensure technical performance requirements are satisfied.

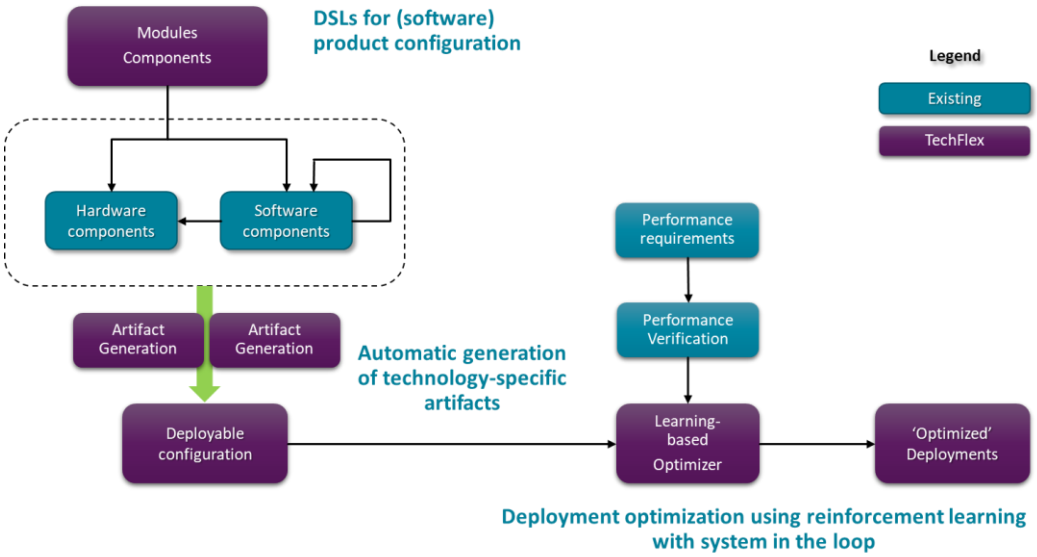

Figure 1: Overview of the TechFlex methodology.

This report primarily addresses the first research question. The second research question considered in a TNO-ESI internal report [3], which presents an initial feasibility study of using reinforcement learning using Monte Carlo Tree Search for the deployment optimization problem. The high-level conclusions from this study are presented in Chapter 4 of this report.

The remainder of this report is structured as follows. Chapter 2 introduces a case study that we will use as an example throughout this report. Chapter 3 then explains our approach to technology-agnostic specification, which allows technology-specific software deployments to be automatically generated. Lastly, Chapter 4 rounds off the report with conclusions and future work.

2 Case Study

The case study in this work is a fictive Meal Delivery System (MDS) that is inspired by a product developed by our industrial partner. An illustration of this system is shown in Figure 2. A customer sends a meal order request to the system. The customer is very hungry and immediately starts walking towards the kitchen after placing the order to get the food as fast as possible. Meal order requests arrive at a Planner service in the system, where a schedule is made and continuously updated to make all customers get their food as fast as possible, subject to availability of limited resources, such as kitchens and delivery bikes. The most recent schedule is continuously sent to a Meal Request Dispatcher that waits for the right time to dispatch a request to start preparing a meal in an available kitchen. Once the meal has been prepared, it is loaded on an allocated delivery bike that tries to reach the customer and deliver the food as fast as possible. Since the time it takes for the bike to deliver the meal to the customer depends on environmental conditions, traffic, and how quickly the customer is travelling, it continuously sends updates about its progress to the Planner, such that it can better estimate when it will be available to deliver the next meal.

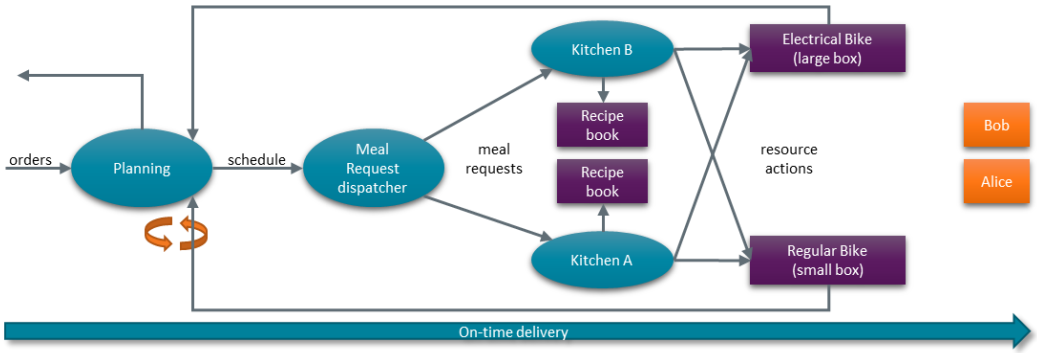

Figure 2: A functional overview of the Meal Delivery System

The Meal Delivery System is an instance of a more general Meal Delivery Product, shown in a simplified feature diagram in Figure 3. The figure shows that the product has two main modules, Data Processing and Meal Delivering. A module is a building block that may in turn comprise other modules, but also components that implement the actual functionality of the system. Using a dashed line, the figure illustrates that the Data Processing module is optional and is not necessarily included in the system. This means that the product can be configured in two distinct variants, the entry level MDS-200 system, which does not have the optional module and its associated functionality, and the premium MDS-600 system that does. This means that we need to be able to configure the product to instantiate both the MDS-200 and the MDS-600 from the set of reusable components. A product with only two variants is trivial by industry standards, but it is sufficient to explain the point of what we are trying to achieve with this research.

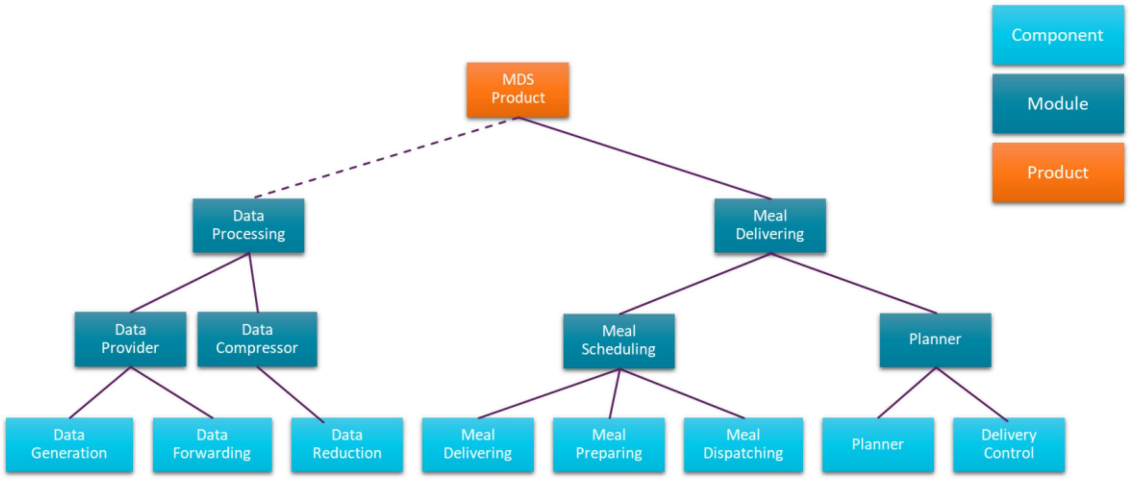

Figure 3: Modules and components in the Meal Delivery Product.

There are other good reasons to integrate a set of components beyond instantiating members of the product family. For example, to create test integrations of a subset of components during development, as shown in Figure 4. The figure shows a possible test configuration of the Meal Scheduling module where two implemented components, Meal Preparing and Meal Dispatching, are tested against a simulator of the Meal Delivery component, which requires access to physical delivery vehicles that are not yet available.

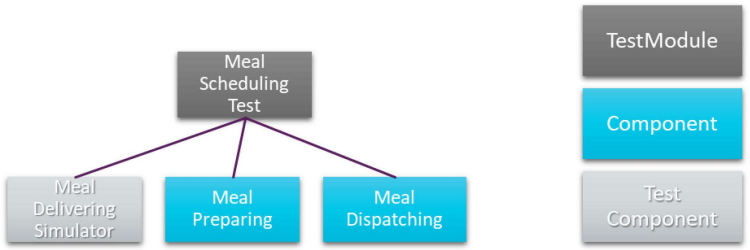

Figure 4: An example test configuration for the Meal Scheduling module.

The challenge in this work is to be able to specify such software integrations in a technology-agnostic manner, such that the specification of a product instance does not change as software technologies evolve. Currently, such integrations are often captured in scripts and Makefiles that are intimately connected to the underlying software technologies and are time consuming and error prone to work with. The challenge of satisfying end-to-end performance requirements after a technology update is addressed in [3].

3 From Specification to Deployment

This chapter describes how challenges related to software variability and evolution are addressed in the project through a model-based approach. First, Section 3.1 presents relevant state-of-the-art, followed by a description of the TechFlex approach in Section 3.2. Next, Section 3.3 presents a software life-cycle model that specifies the dependencies between software artifacts in different life cycle phases. Section 3.4 presents a family of DSLs that specify software configurations in a technology-agnostic manner, along with a set of generators that automatically create software deployments from those configurations for specific software technologies. Lastly, Section 3.5 briefly discusses an evaluation of the approach using the Meal Delivery System from Chapter 2 and an industry case study.

3.1 State-of-the-art

System diversity, or variability, is a recognized challenge across industries. A survey covering 18 companies in different industries offering B2B products shows that how variability is managed depends on the complexity of the configuration challenge [4]. Companies with limited variability often use spreadsheets to model variants. This approach is easy to use, but overview is quickly lost when the number of variants increases, resulting in large and complex spreadsheets. With this approach, it is furthermore difficult to express constraints between features and validate that a particular configuration is a valid member of the product family. Companies with more complex configurations often rely on formal models, which have been shown to capture knowledge better and facilitate conversations with domain experts. Two types of formal models are used to capture variability. Different types of *feature models*, e.g. Product Variant Master (PVM), sometimes complemented by Class Responsibility Collaboration (CRC) cards that detail the individual object classes and Unified Modelling Language (UML) class diagrams, enriched with Object Constraint Language (OCL) constraints. PMV is considered easier for non-experts to use, but UML is richer and part of a standard [5]. According to a somewhat dated survey [6], there are several tools available for feature modelling. Some of them are academic, some commercial, and others available in both forms.

While feature models are specifically designed to express variability in systems, they are not the only option. The benefits of using domain-specific languages (DSLs) to address variability in the context of product line engineering is discussed in [7]. The authors state that basic feature models are context-free grammars without recursion, which makes them limited in terms of expressivity. Often, there is a need for recursion to allow multiple instantiations of an element, references to establish links between them, and attributes to increase the configuration space. Some of these features are available in more expressive types of feature models. The authors argue that DSLs fill a gap between feature models and general-purpose programming languages, keeping the separation between the problem domain, “the what”, and the solution domain, “the how”. DSLs are furthermore able to specify algorithmic behavior and unbounded configuration spaces. Because the syntax of a DSL can be tailored to the particular domain, it also makes it more understandable to non-

programmers, which may improve communication across disciplines and enable systems engineers to work more independently from software engineers.

It is also possible to combine feature models and DSLs. Three categories of combinations are discussed in [7]:

1. A feature model is used to specify variations in a model.
2. A feature model is used to specify variations in model transformations.
3. Components are specified/implemented using DSLs and feature models are used to combine them into products.

There are multiple ways in which formal models that express variability can be connected to the system development process. The high-tech industry is showing increasing interest in Model-based Systems Engineering (MBSE) [1], [8]. In terms of formalism, MBSE is predominantly relying on SysML, a general modelling language which originated as an extension of UML to cover the domain of systems engineering. SysML v1 does currently not define any specific constructs or concepts for variation modelling [9]. For this reason, UML profiles, such as VAriation MOdeling with SysML (VAMOS), have been developed to extend the language to allow product variations to be defined. In contrast, SysML v2 is developed independently from UML and directly supports concepts for variation management [10]. Currently, this functionality is available for beta testing within commercial tools. There are also third-party tools for variation management. For example, pure::variants [11] is a prominently used tool in the Dutch high-tech equipment industry. It integrates with many known tools for MBSE, MBD, Requirements management, and Test Management tools.

Another direction is to use formal models for variability management in the implementation of *product configurators*. These are software-based expert systems, tools that support the user in the creation of product specifications by restricting how predefined entities and their properties may be combined [12]. An empirical study on the business impact of product configurators with focus on how business activities are affected, challenges in design, development, and maintenance, and barriers for adoption, is provided in [13]. In this an analysis is provided based on a survey with 64 respondents across domains including telecom, computer, and industrial machinery. The results show that product configurators are often used by sales people, customers, designers, and planners to specify or configure products. They are typically integrated with other systems to generate e.g. documentation, invoices, bill of materials, and drawings. In that sense, product configurators can be seen as an example of *model-based engineering*, where models are used as a key source of information and as a source for generating relevant artifacts, and share many characteristics with this engineering approach [14], [15], [16], [17].

Although there are substantial benefits of product configurators, empirical results [13] suggest their introduction increase rather than reduce the number of employees in the organization. Normal change management practices apply when introducing a product configurator. You need long-term commitment at all levels, and you need to watch out for people whose value in the organization becomes reduced, or whose job becomes redundant. This is also consistent with existing literature on the introduction of model-based engineering methodologies [15], [16]. A key challenge with their introduction is the availability of IT people to design, develop, and maintain the configurator, especially as the system evolves. If these people are brought in externally, they do not have sufficient domain knowledge [13].

Different strategies for developing product configurators based on published case studies from engineering-to-order companies are identified and discussed in [5]. These strategies

illustrate different workflows for sharing, formalizing, and implementing product information involving roles such as product experts, knowledge representation experts, and configurator software experts. The main differences between the different strategies are the extent to which the different tasks and roles are distinguished and merged. The strengths and weaknesses of each strategy are discussed along seven dimensions and guidelines are given for when each strategy may be useful. For complex configuration projects, it is recommended to have a knowledge representation that is separate from the software implementation to better communicate with product experts. This allows product rules to be validated using the knowledge representation instead of using the implementation. For simpler cases, costs can be reduced if product information is encoded directly in the configuration system. Another factor that impacts the choice of strategy is the availability of skilled people and whether it is an option to have the same people doing knowledge representation and implementation. Application of product configurators in engineering-oriented companies has resulted in many benefits, e.g. shorter lead times, improved certainty of delivery, fewer resources for specification, improved quality of specifications, optimized products, preservation of knowledge, and less time for training new employees [5].

3.2 The TechFlex Approach

We continue by positioning the approach to technology-agnostic specification of software variability in this project, with respect to the existing body of work, presented in Section 3.1. The TechFlex approach uses a family of DSLs as the formalism to express variability in software configurations in a technology-agnostic manner. The choice of a DSL is motivated partly by the need to express how systems are recursively built up from modules and components, which excludes some basic approaches to feature modelling as previously mentioned in Chapter 2. The decision was also based on which technologies were available and familiar and in use within the organization of our industry partner, which increases technology acceptance and makes it easier to successfully embed the developed methodology in the organization and create industry impact. It also addresses challenges related to introduction of product configurators, previously addressed in Section 3.1, such as availability of IT people for development and maintenance. The family of DSLs is essentially a software product configurator that is connected to a set of technology-specific generators that automatically generate software deployments from a single source of truth, a best practice in model-based engineering [17]. The separation of languages for specification and generators for implementation, means that our approach has a separate knowledge representation, making it suitable for complex configuration projects [5]. While the proposed methodology is quite general, tailoring it to different organizations may involve revisiting technology choices, whether that is using different tools, like pure::variants, as a basis for the approach, or a different language workbench for making DSLs.

3.3 Software Life-cycle Meta-model

Before creating a software product configurator that addresses challenges related to variability and evolution, it is important to understand the artifacts involved in the software life-cycle, as well as their dependencies. Software is specified in either text or a more or less formal model, and designed using either simple descriptive drawings or in more formal structural and behavioral models, e.g. using state diagrams or message sequence charts. It is implemented in a certain programming language using source files and header files. Then, it is generated into binary forms for a certain hardware/software platform and packed in

deployable entities like executables or container files. Finally, it is deployed to a certain platform, ranging from a simple real-time kernel on a single embedded processor to a complex deployment and monitoring system like Kubernetes running on a large Linux cluster. The life-cycle captures all types of elements, such as component specifications, interfaces, source code, runnable units, containers, and deployment files, in the different life-cycle stages of the software and the relations between them.

Historically, the different phases as described above are not always well separated and technology choices like language constructs or platform specifics are mixed already in the specification and design phases. A software life-cycle meta-model is hence helpful in analyzing what technology changes impact what phase in the software development and generation. This allows a cleaner separation between the phases in which technology choices can be pushed to the later phases of development, supporting less disruptive technology evolution without the need of specification or design updates. Such a model allows changes to only propagate downwards, i.e. a change in implementation or deployment does not affect the specification.

A software life-cycle meta-model, inspired by [18], was created for the Thales environment, following an approach that has previously been successfully applied in the high-tech equipment industry in a research project with ASML [19], [20]. The meta-model was created using the Eclipse Modelling Framework (EMF) and considered four software life-cycle stages relevant to this work. While the model itself is too large and complex to show in the report, we provide a list of the stages, along with examples of key artifacts that were identified in each stage and an analyses of their technology dependence. This is visualized in Figure 5.

1. **Specified:** Artifacts that contribute to the specification, such as specifications of components and modules, along with their required and provided interfaces and capabilities. The artifacts in this stage can be technology-agnostic.

2. **Implemented:** Artifacts that are part of the implementation, such as source code and header files. The artifacts in this stage depend on the implementation technology, e.g. the programming language.

3. **Generated:** Artifacts that are generated from the implementation artifacts, such as bundles, applications, and runnable units. These artifacts depend on the platform and containerization technology.

4. **Instantiated:** Artifacts relevant to deployment of the software, e.g. deployment files. These artifacts depend on the particular deployment technology, e.g. Kubernetes.

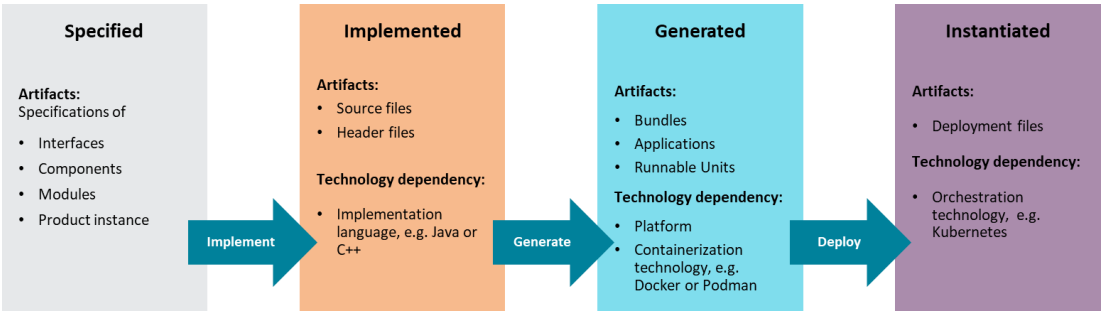

Figure 5: The four stages of the software life-cycle model, along with key artifacts and technology dependencies.

This clear separation of phases isolates technology dependencies (implementation, platform and deployment technologies) in specific stages. The specification of the product components, their variability and instantiation into a specific product is now independent of technological evolution, the impact of which is moved to subsequent phases in which most artefacts can be generated (compiled/build) automatically.

In the remainder of this document, we present a set of DSL's describing the specification and variation of components in the specification phase. In addition they describe the automation (generation) of the transitions between the phases. using technology-specific generators.

3.4 Domain-specific Languages

Building on the software life-cycle meta-model that describes the elements in the different phases, we define a set of DSLs that allow us to specify, structure and create concrete instances of specification models. The main idea is that these DSLs are *technology-agnostic* and that DSL instances, describing different elements of a software product instance, do not require updates if, e.g., the implementation technology or deployment technology is updated.

These DSLs are accompanied by a set of generators that perform phase-to-phase transformations. Like the specification models, the generators are abstracted by a set of DSL's, but their implementation is *technology-specific*. These generators are used to automatically create relevant development artifacts, reducing effort, in particular in light of increasingly frequent updates. The benefit of encapsulating technology-specific aspects in generators in this way is that it addresses technology changes at the level of the product family instead of the individual variant, improving scalability as the number of variants increases.

Note that while the approach makes software configurations independent from changes in software technologies, they are sensitive to changes in the DSLs themselves. If the DSLs evolve to include new concepts, it may become necessary to co-evolve the configuration models to ensure compatibility. This challenge and methods to address it is discussed in [21].

Figure 6 shows the family of DSLs and generators related to a more detailed view of the software lifecycle model

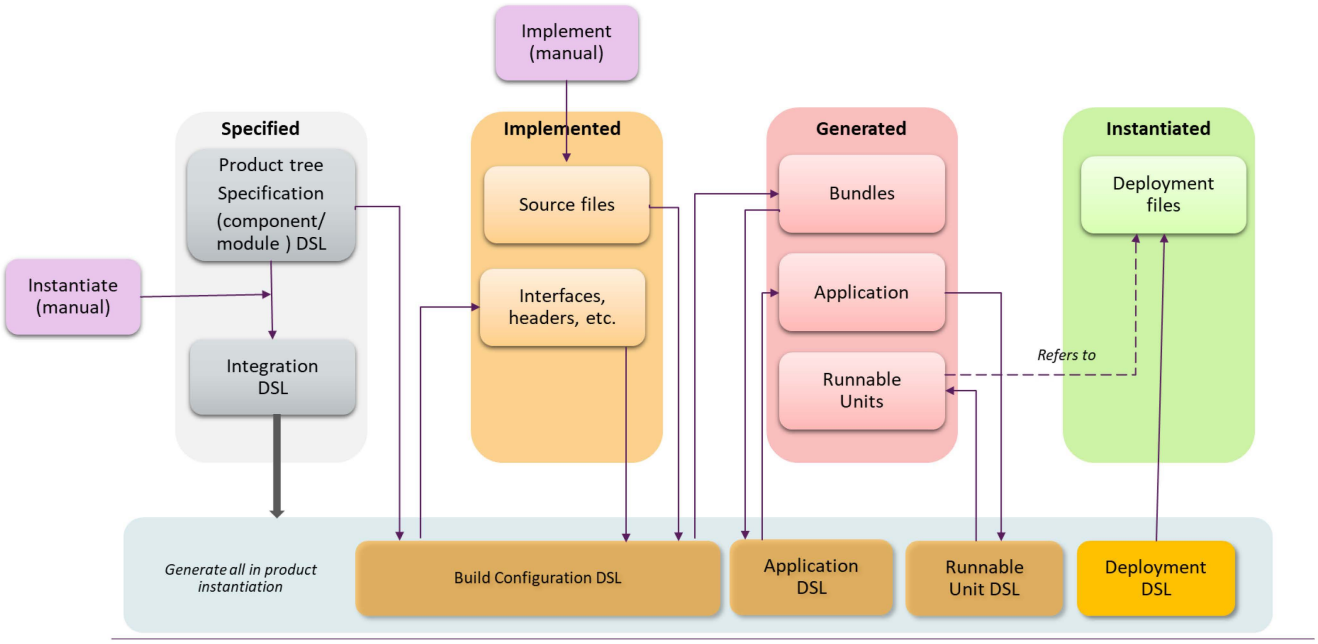

Figure 6: Detailed view software lifecycle phases, DSL's and generation.

We continue by providing a simplified high-level overview of these DSLs to communicate the key concepts. The gray boxes (left) correspond to the specified stage of the software life-cycle meta-model previously presented in Section 3.3. Metaphorically, this perspective specifies the Lego pieces that are available to build with, as well as the instructions for how to put them together to create a new instance. Component DSL specifies the components and through Interface DSL (not shown in the above picture) the interfaces that components and modules use to communicate with each other. Note that Interface DSL and Component DSL were not developed as a part of this project, but have been previously developed and used by our industry partner. In that sense, our work is extending an existing DSL ecosystem and does not introduce new technology, making it easier to transfer the results. Module DSL is an extension that expresses how modules are recursively built up from other modules and components, allowing a structured view on the sea of components and expressing variation points. Integration DSL specifies which components and modules come together in a particular software configuration, corresponding e.g. to a new product instance or a test integration, as previously explained in Chapter 2.

To further torture the metaphor, the phases after specification are all about making sure the Lego pieces are correctly manufactured and ready for assembly. This corresponds to the orange boxes in the bottom of Figure 6, which make sure that all components of the software specified by the Integration DSL are actually built and ready for deployment. First, the Build Configuration DSL has a link to an instance of the Component DSL, since it will generate the implementation for a specified component. It specifies how to create a build project related to a particular component and specifies where the corresponding implementation (source code) can be found. The associated technology-specific generators generate and expose Java bundles and Fat jars for Java development, respectively. Next, the Application DSL specifies an application as a set of related components available in the exposed bundles. Again, this is done in a technology-agnostic way by abstractly referring to the artifact where the bundles can be found. The associated technology-specific generators

generate the build source for an application that when executed builds the application according to the particular implementation technology, i.e. as an Apache Celix [22] application for C++ development or a fat jar for Java development. The last part of the implemented stage is the Runnable Unit DSL, which specifies which applications should be combined into a runnable unit and always be deployed together. Based on this specification, the technology-specific generators create the build directives of a container image for particular containerization technologies, such as Docker or Podman.

The final DSL, Deployment DSL, corresponds to the deployed stage of the software life-cycle model. This DSL defines which runnable units (in the specified software configuration) should be deployed on which compute node. The technology-specific generators generate deployment files for the particular deployment technology, such as Kubernetes.

An overview of the workflow using this family of DSLs is shown in Figure 7.

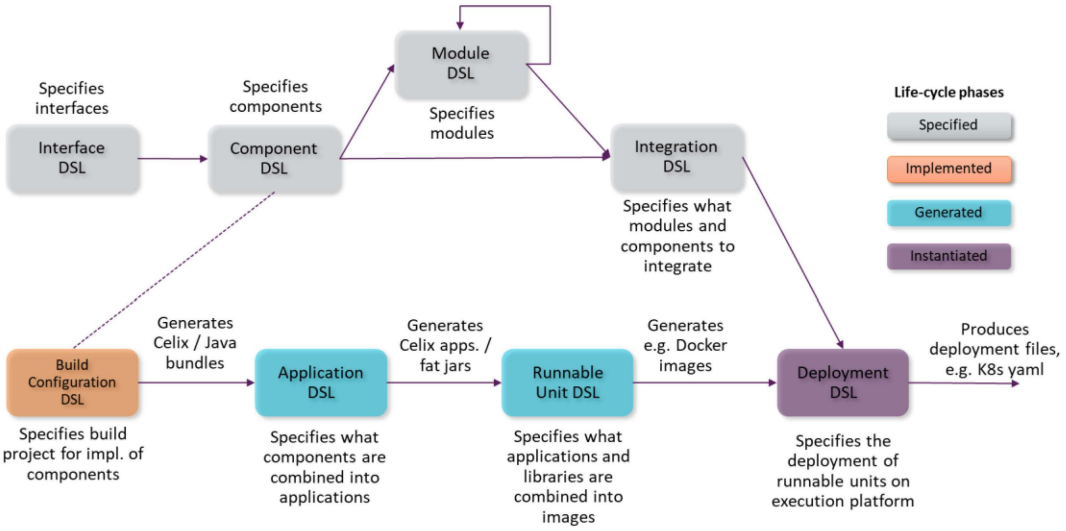

Figure 7: The family of technology-agnostic DSLs and their dependencies.

The proposed DSL-based approach addresses the challenges related to both system variability and system evolution, previously described in Chapter 1, in a model-based manner. The system variability problem is addressed since new software configurations can easily be specified as instances of technology-agnostic DSLs that do not require updates as software technologies change, minimizing manual intervention and effort when new variants are specified and maintained. The software evolvability problem related to deprecation of software technologies during the life-time of the system is addressed through the technology-specific generators that allows a change in implementation technology from Java to C++ or from Docker and Kubernetes to alternative containerization and orchestration technologies to be handled by phasing in and phasing out technology-specific generators. This way of improving portability in terms of technology through model-based engineering in general and domain-specific languages in particular is recognized in surveys [23], [24] and case studies [25].

Next, we provide more information about the family of DSLs created as a part of this work. In the following sections, we demonstrate the DSLs based on a running example with our Meal Delivery System, previously introduced in Chapter 2.

3.4.1 Interface DSL

The Interface DSL builds on the Apache Avro [26] Interface Definition Language (IDL) and defines data structures and function signatures. Models in this DSL are used by the Component DSL and Module DSL by referring via Component/Module DSL's declarations *require service <avdl reference>* and *provide service <avdl reference>.*

In the example in Figure 8, we see OrderApi protocol declaration containing *OrderRequest* data structure declaration and *SubmitOrderRequest* signature declaration. Every component referring to OrderApi would use *require service OrderApi* and/or provide service OrderApi. You can also see annotations (@version, @behavior, and @cxx-namespace), which are hints for the generators. Some of these hints, such as @cxx-namespace is clearly technology dependent. This is because Interface DSL was developed by our industry partner prior to this project. A solution to this could be to e.g. introduce a technology-agnostic annotation for declaring namespaces that are independent of a particular programming language.

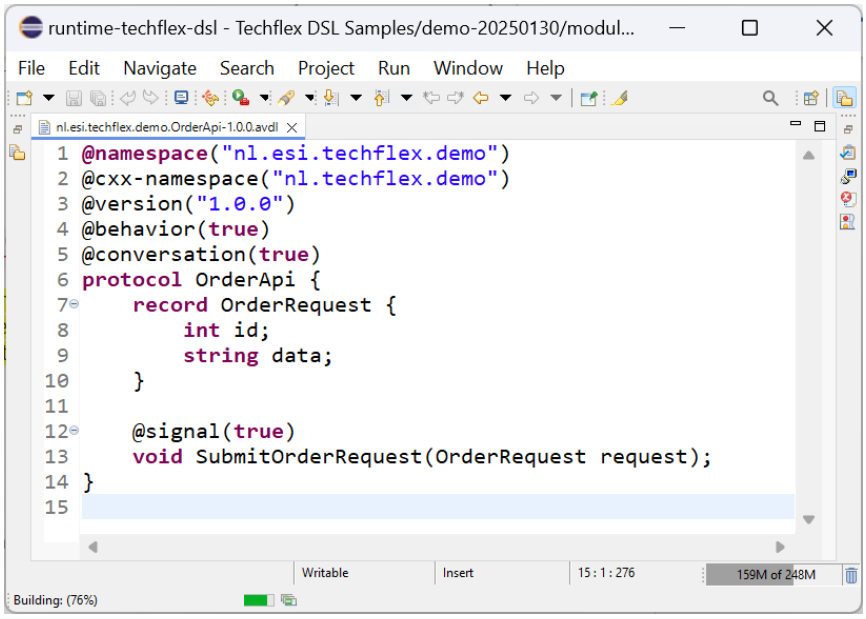

Figure 8: An example service defined in Interface DSL.

3.4.2 Component DSL

The Component DSL describes the smallest decomposition entity within a system. It describes which messages can be sent and received by a component, and what the structure of these messages looks like.

In the example in Figure 9, we see the specification of the Planner component. With the *provide service* OrderApi declaration, we define which messages the component can receive. Similarly, with the declarations *require service* DispatchApi and NotificationApi, we define which messages it can send.

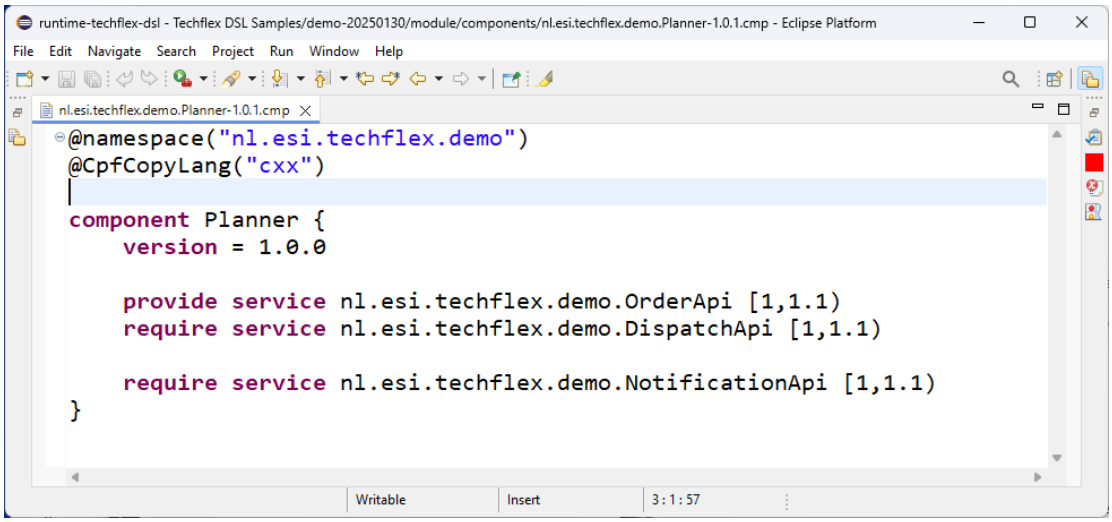

Figure 9: An example component defined in Component DSL.

3.4.3 Module DSL

With Module DSL, one can define a full product tree from system level, down to individual components through hierarchical (recursive) composition. The need for this was previously stated in the context of our case study in Section 2 and was one of the reasons for choosing DSLs as the approach to specify software configurations in Section 3.2. The central element of this recursive model is the assembly, which can be either a module (an assembly that is further decomposed) or a (leaf element) component. The assemblies are specified by referring to individual provided and/or required services. Assemblies need be matched to modules or components with the matching interfaces. In our current proof-of-concept implementation, this matching is hard-coded, but will be replaced with a dynamic resolver in the future.

In the example in Figure 10, we see the MealProcessing module, which refers to a component/module providing MealProcessingCapability. It also refers to three assemblies: 1) a dispatching assembly that provides DispatchApi and requires PrepareApi, 2) one responsible for preparation that provides PrepareApi and requires DeliverApi, and 3) an delivery assembly providing DeliverApi and requiring DeliveryNotificationApi.

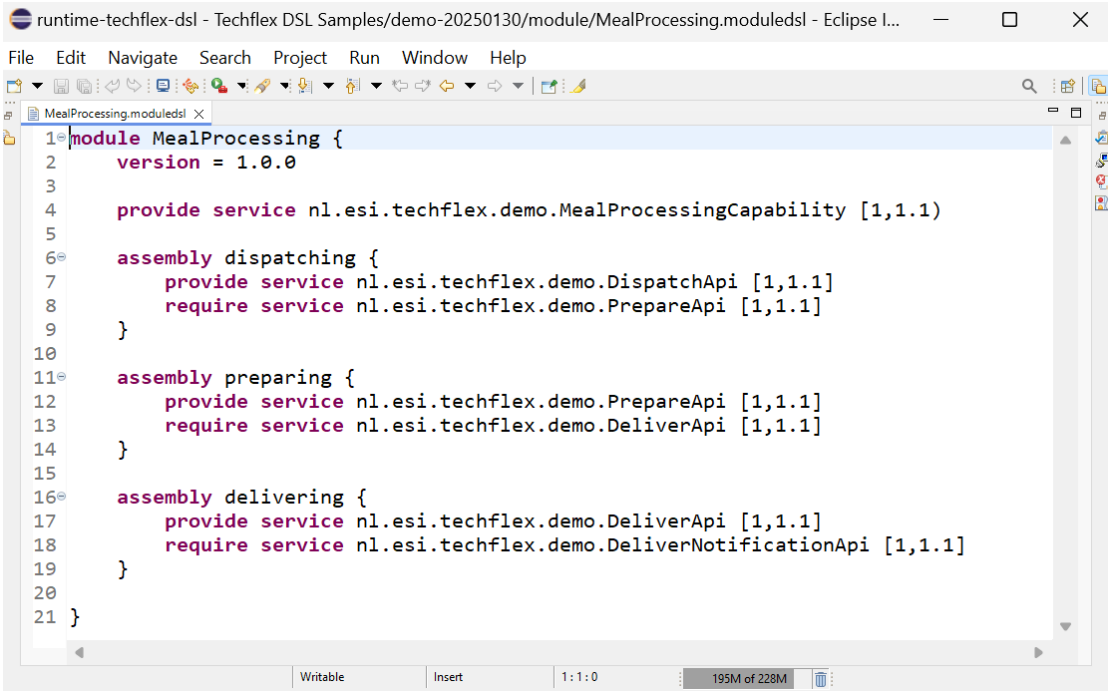

Figure 10: An example instance of Module DSL.

3.4.4 Integration DSL

The Integration DSL defines which components are part of a particular software configuration, e.g. corresponding to a particular product instance or a test integration. In the example in Figure 11, we see an integration named MDS200Int, comprising five components, DeliveryControl, Planner, MealDispatching, MealPreparing and MealDelivering, respectively. In our current proof-of-concept implementation, it is not possible to include Modules in Integration DSL. This will be added as a part of future work.

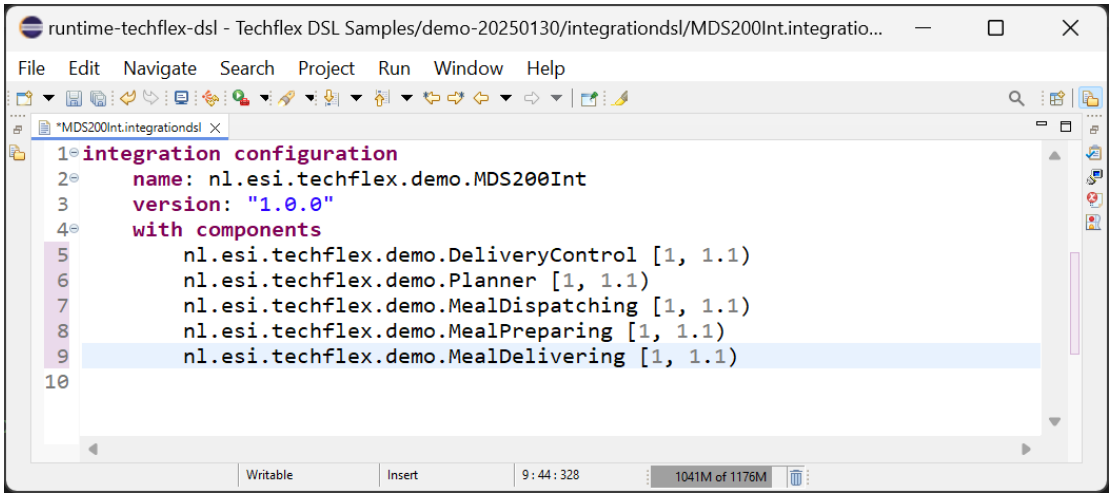

Figure 11: An example instance of Integration DSL.

3.4.5 Build Configuration DSL

The Build Configuration DSL defines a build project containing a set of components and dependencies, e.g. on specific libraries, which we are going to implement in a particular programming language. The declarations of dependencies are technology-agnostic, but refer to a technology-specific artifact, such as jar files, OS native shared libraries, or Python modules, identified by a combination of groupId, artifactId and version. This type of technology-agnostic references are also used in other DSLs in this ecosystem.

With a concrete implementation generator for a certain programming language and/or framework (e.g. Spring Boot [27] or Django [28]), one creates all related resources (e.g. source files and configuration files) needed for building (compiling, linking, and packing) a language-specific artifact, such as shared lib, jar file, or Python modules.

In the proof-of-concept implementation, we created a generator to create a build project with a predefined directory structure. It generates configuration files like pom.xml, CMakeLists and defined dependencies and final generator C++ *.h, *cpp files related to the Component/Interface DSL declarations. This will be used as a starting point for developers to implement their business logic for the components and run the generated technology-specific build scripts, in our case *mvn package* which builds a C++ Celix bundle.

The example in Figure 12 specifies a build configuration of the artifact *planning* containing the components *Planner* and *PlannerSimulator* which provides the protocols *OrderApi* and *DispatchApi*. When the generated build project runs its build scripts (e.g. mvn package), it produces an artifact (e.g. Celix Bundle) referenced by groupId: “nl.esi.techflex.demo”, artifactId: “planning” and version “1.0.0-SNAPSHOT”.

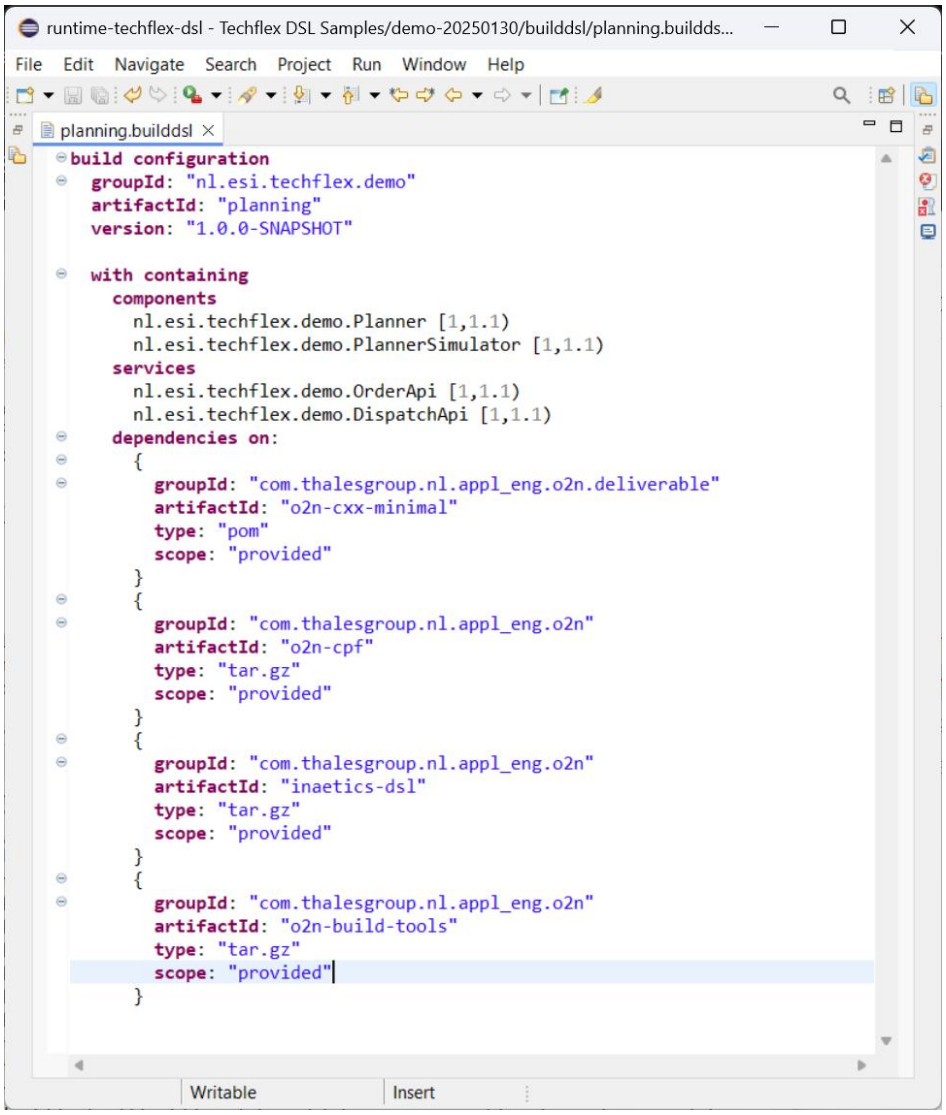

Figure 12: An example instance of Build Configuration DSL.

3.4.6 Application DSL

The Application DSL is used to compose a concrete application based on the defined components and dependencies. It contains the components we want in the application and the dependencies, where we can find the technological-specific implementation of those components. The generated application artifact can be referenced by its groupId, artifactId and version.

With a concrete implementation of a generator, we can create a native executable that can be executed. The generator knows how to link all referenced artifacts in the dependencies to a native application. In our proof-of-concept, we implemented a generator for making C++ Celix applications.

In the example in Figure 13, we generate an application called order-processing-app, which contains two components, the *Planner* and the *DeliveryControl*.

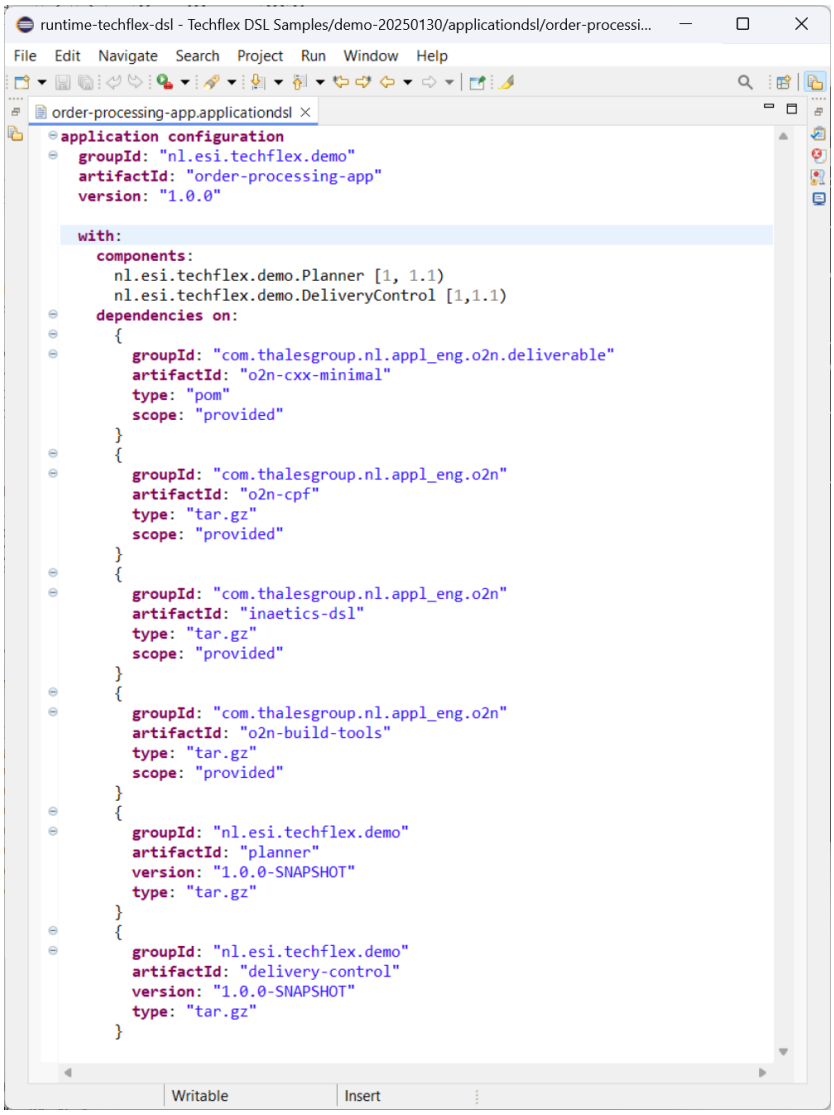

Figure 13: An example of Application DSL.

3.4.7 Runnable Unit DSL

The Runnable Unit DSL defines which application artifacts defined by Application DSL should be bundled, executed and/or deployed together.

In our proof of concept, we implemented a generator that created container images. The example in Figure 14 specifies a runnable unit called order-processing-unit that comprises two applications, order-processing-app and meal-processing-app, respectively. The generated image artifact can be referred via its groupId, artifactId and version.

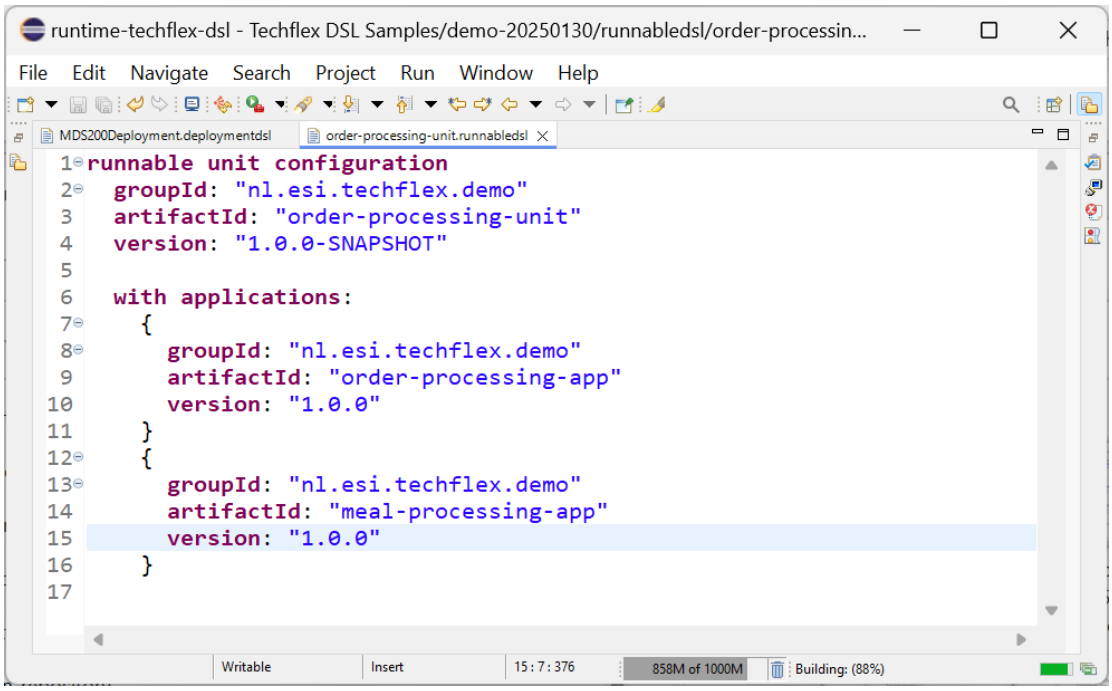

Figure 14: An example of Runnable Unit DSL.

3.4.8 Deployment DSL

The Deployment DSL defines on which hosts a set of runnable units should be deployed. The hosts are defined using a filter mechanism that selects out of a total set of available hosts.

With a concrete implementation of generator, you can create all needed scripts, configs and resources needed for your deployment. In our proof of concept, we implemented a generator that creates deployment scripts for Kubernetes. The example in Figure 15 defines a deployment configuration called MDS600Deployment that deploys two runnable units, order-processing-unit and meal-processing-unit, on a host with name archive01.tsn.tno.nl.

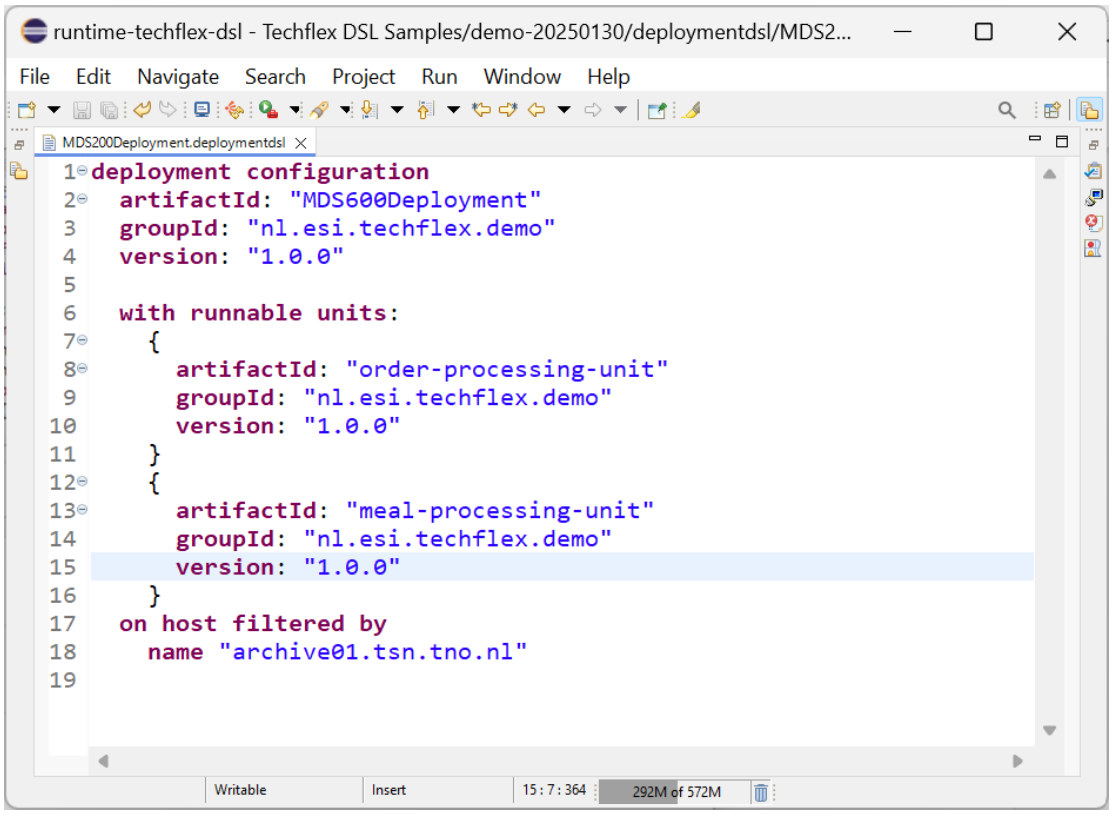

Figure 15: An example of Deployment DSL.

3.5 Evaluation

The model-based approach to technology-agnostic specification of software configurations was evaluated by applying it to the Meal Delivery System, previously introduced in Chapter 2. This evaluation revealed that the family of DSLs was sufficiently expressive to specify the relevant artifacts in each life-cycle phase. From these specifications, it was also possible to automatically generate working software deployments using the technology-specific generators. In addition, our industry partner applied the approach on a confidential industry case study from the defense domain. Similarly to our findings, the DSL was found to be sufficiently expressive to cover the needs of this case study.

4 Conclusions

This chapter concludes the report by first answering the research question in Section 4.1, followed by a discussion about future work in Section 4.2.

4.1 Answers to Research Questions

RQ1) To what extent can we generate efficient software deployments for different technologies from a generic software configuration model?

- To what extent can the generic software configuration model be decoupled from the deployment technology?
- Can it be completely decoupled while still generating efficient deployments, or only encapsulated in a separate technology-specific model?

This work has demonstrated that it is possible to fully specify software configurations and deployments in a technology-agnostic way using a family of DSLs (see Section 3.4). Software configurations are defined using an Integration DSL that refers to modules (Module DSL) and components (Component DSL), which in turn request and provide services through their interfaces (Interface DSL).

There are also a set of technology-agnostic DSLs that define how components are implemented (Build Configuration DSL) and combined into applications (Application DSL) and runnable units (Runnable Unit DSL) that are deployed on the compute nodes (Deployment DSL). These DSLs are paired with technology-specific generators that encapsulate all technology-specific aspects, such as whether a component implementation is provided in Java or C++, and the containerization and orchestration technology. This means that only generators need to be added and removed as software technologies evolve, while the DSL specifications are unaffected. Given the same DSL specifications, new implementations or deployments can then be automatically generated for new technologies.

We validated our work on a simple case study, the Meal Delivery System introduced in Chapter 2. In addition, our industrial partner has validated the expressivity of the DSLs on a confidential case study in their domain. We have concluded that the DSLs are sufficiently expressive to cover these case studies and that the proposed approach addresses the problem of technology-agnostic specification in a good way. The efficiency of the generated deployments was not evaluated during the project. Instead, generated deployments were optimized in a separate stage, as described next.

RQ2) How can we efficiently optimize software deployment(s) to satisfy performance requirements for a particular product configuration, load, and deployment technology?

- What are the benefits and drawbacks of performance prediction vs. performance evaluation?

This question is investigated in a TNO-ESI Internal report [3] and only the main conclusions are presented here. There are many ways to optimize software deployments to satisfy

performance requirements. We considered different ways of navigating design spaces, including exact approaches, (meta-)heuristic approaches, and learning-based approaches, as well as methods for evaluating selected points in that design space, such as evaluation on the real system, analytical models, and AI-based methods.

The selected approach navigates the design-space a learning-based approach, more specifically Reinforcement Learning using Monte Carlo Tree Search and performs evaluations of selected mappings on the real system. Reinforcement Learning was chosen as it is an established approach to solve deployment optimization problems in literature. Selected design points are evaluated on the real system, which is significantly more time-consuming than using performance prediction using analytical models, but provides the strongest guarantees that the performance requirements will be satisfied.

A simple proof-of-concept implementation was developed clearly separates the optimization and evaluation approaches and can hence be used either with the real system, or a model thereof. Due to limited time, an elaborate evaluation of the optimization approach was not possible during the project, but initial feasibility was demonstrated using our proof-of-concept implementation in the context of the Meal Delivery System presented in Chapter 2. Further research is required to assess the effectiveness and scalability of the approach. While it is certainly possible to use reinforcement learning to optimize deployments at design time, much like a meta-heuristic, it is our impression that the approach is a better conceptual fit for run-time problems. One reason is that reinforcement learning is not looking for solutions to a problem, but for a policy to guide sequential decision making. The solution itself is a secondary result of applying the optimized policy to a set of sequential decisions. Another reason is that the run-time case allows continuous learning in the operational system, allowing the policy to adapt to changing system dynamics.

4.2 Future Work

Future work involves creating additional abstraction levels to add automation and further reduce the impact of technology changes. This work proposed a model-based methodology for making technology-agnostic specification of components and their integrations, and technology-specific generation of artifacts. While this approach makes specifications resilient to changes in technology, new generators may have to be introduced, or existing generators significantly modified, as a result of a technology change. Future work could address this by defining technology-agnostic APIs to further structure the generators and the underlying build environment.

Another direction involves adding additional levels of abstraction and automation on top of the existing approach to further reduce manual effort in creating software configurations that satisfy functional and non-functional requirements. In particular, it could be possible to specify the required capabilities of a software configuration and automatically synthesize the set of components and modules to integrate based on the capabilities they require and provide. In terms of the work in this project, this could involve automatically generating instances of Integration DSL based on a set of required capabilities. Satisfying performance requirements in this context becomes even more challenging, as many different software configurations may be considered by the synthesis, significantly increasing the design space. The current approach of optimizing deployments on actual hardware is not expected to be able to address this case. However, if synthesis is linked to a model-based approach to predict performance of key systems flows of a software configuration for a particular mapping, e.g. based on [29], it could be possible to also automatically generate an instance

of Deployment DSL, resulting in a deployment that directly satisfies performance requirements. In this case, the current RL-based approach to deployment optimization on the real system can be used after synthesis for final parameter optimization and verification. This direction will be investigated in a follow-up project.

References

[1] T. Hendriks and S. Acur, “Vision and Outlook for Systems Architecting and Systems Engineering in the High-Tech Equipment Industry,” TNO 2024 R10542, 25 2024. Accessed: Oct. 21, 2024. [Online]. Available: https://repository.tno.nl/SingleDoc?find=UID%2091ca077f-403c-4589-9d04-da0695f463e6

[2] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, “Microservices: The Journey So Far and Challenges Ahead,” *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018, doi: 10.1109/MS.2018.2141039.

[3] B. Akesson and I. Armengol Thijs, “Deployment Optimization using Reinforcement Learning.” TNO-ESI, 2025.

[4] L. Hvam, M. Bonev, A. Haug, and N. H. Mortensen, “The Use of Modelling Methods for Product Configuration in Industrial Applications,” in *Proceedings of the 7th World Conference on Mass Customization, Personalization, and Co-Creation (MCPC 2014), Aalborg, Denmark, February 4th - 7th, 2014*, T. D. Brunoe, K. Nielsen, K. A. Joergensen, and S. B. Taps, Eds., Cham: Springer International Publishing, 2014, pp. 529–539. doi: 10.1007/978-3-319-04271-8_44.

[5] A. Haug, L. Hvam, and N. H. Mortensen, “Definition and evaluation of product configurator development strategies,” *Computers in Industry*, vol. 63, no. 5, pp. 471–481, Jun. 2012, doi: 10.1016/j.compind.2012.02.001.

[6] M. El Dammagh and O. De Troyer, “Feature Modeling Tools: Evaluation and Lessons Learned,” in *Advances in Conceptual Modeling. Recent Developments and New Directions*, vol. 6999, O. De Troyer, C. Bauzer Medeiros, R. Billen, P. Hallot, A. Simitsis, and H. Van Mingroot, Eds., in Lecture Notes in Computer Science, vol. 6999. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 120–129. doi: 10.1007/978-3-642-24574-9_17.

[7] M. Voelter and E. Visser, “Product Line Engineering Using Domain-Specific Languages,” in *2011 15th International Software Product Line Conference*, Munich, Germany: IEEE, Aug. 2011, pp. 70–79. doi: 10.1109/SPLC.2011.25.

[8] B. Pronk, “MBSE Study: Conceptual framework on Variation Modelling,” TNO, 2023–10215.

[9] T. Weilkiens, *Variant Modeling With SysML*. MBSE4U. Accessed: Oct. 25, 2024. [Online]. Available: https://mbse4u.com/books/variant-modeling-with-sysml/

[10] “About the OMG System Modeling Language Specification Version 2.0 beta 2.” Accessed: May 21, 2025. [Online]. Available: https://www.omg.org/spec/SysML

[11] “pure::variants.” Accessed: Oct. 25, 2024. [Online]. Available: https://www.pure-systems.com/purevariants

[12] A. Haug, *Representation of Industrial Knowledge - as a Basis for Developing and Maintaning Product Configurators*. 2008.

[13] L. Zhang, P. Helo, A. Kumar, and X. You, “An empirical study on product configurators’ application: Implications, challenges, and opportunities,” presented at the Configuration Workshop, 2015, pp. 5--10.

[14] B. Akesson, J. Hooman, R. Dekker, W. Ekkelkamp, and B. Stottelaar, “Pain-mitigation Techniques for Model-based Engineering using Domain-specific Languages,” *Proc. Special Session on Model Management And Analytics (MOMA3N)*, 2018, Accessed: Jan. 09, 2025. [Online]. Available: https://www.scitepress.org/Papers/2018/67497/67497.pdf

[15] B. Akesson, J. Hooman, R. Dekker, W. Ekkelkamp, and J. Sleuters, “Pains & Gains when Transitioning to Model-based Engineering using Domain-specific Languages,” TNO, Jan. 2019.
[16] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, “Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice,” *Softw Syst Model*, vol. 17, no. 1, pp. 91–113, Feb. 2018, doi: 10.1007/s10270-016-0523-3.
[17] W. Oortwijn *et al.*, “Vision and Outlook for System Evolution and Diversity,” TNO 2024 R10716, Feb. 2025. [Online]. Available: https://publications.tno.nl/publication/34643767/xtSUpmKv/TNO-2024-R10716.pdf
[18] H. Koziolek, “Performance evaluation of component-based software systems: A survey,” *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, Aug. 2010, doi: 10.1016/j.peva.2009.07.007.
[19] “PPS: From Methodology to Application in Industry : A method for analyzing performance on complex Cyber Physical Systems.” Accessed: Jan. 09, 2025. [Online]. Available: https://repository.tno.nl/SingleDoc?docId=58556
[20] K. Triantafyllidis, S. Niknam, Y. Blankenstein, and J. Hegge, “Platform Performance Suite (PPS): A framework for performance analysis & diagnosis of complex cyber-physical systems,” in *16th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2025.
[21] B. Akesson, J. Hooman, J. Sleuters, and A. Yankov, “Reducing design time and promoting evolvability using Domain-Specific Languages in an industrial context,” in *Model Management and Analytics for Large Scale Systems*, B. Tekinerdogan, Ö. Babur, L. Cleophas, M. van den Brand, and M. Akşit, Eds., Academic Press, 2020, pp. 245–272. doi: 10.1016/B978-0-12-816649-9.00020-X.
[22] “Apache Celix.” Accessed: Jan. 09, 2025. [Online]. Available: https://celix.apache.org/
[23] M. Torchiano, F. Tomassetti, F. Ricca, A. Tiso, and G. Reggio, “Relevance, benefits, and problems of software modelling and model driven techniques—A survey in the Italian industry,” *Journal of Systems and Software*, vol. 86, no. 8, pp. 2110–2126, Aug. 2013, doi: 10.1016/j.jss.2013.03.084.
[24] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, “Empirical assessment of MDE in industry,” in *Proceedings of the 33rd International Conference on Software Engineering*, in ICSE ’11. New York, NY, USA: Association for Computing Machinery, May 2011, pp. 471–480. doi: 10.1145/1985793.1985858.
[25] N. Mellegård, A. Ferwerda, K. Lind, R. Heldal, and M. R. V. Chaudron, “Impact of Introducing Domain-Specific Modelling in Software Maintenance: An Industrial Case Study,” *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 245–260, Mar. 2016, doi: 10.1109/TSE.2015.2479221.
[26] “Apache Avro,” Apache Avro. Accessed: Mar. 09, 2025. [Online]. Available: https://avro.apache.org/
[27] “Spring Boot,” Spring Boot. Accessed: Jul. 22, 2025. [Online]. Available: https://spring.io/projects/spring-boot
[28] “Django,” Django Project. Accessed: Jul. 22, 2025. [Online]. Available: https://www.djangoproject.com/
[29] M. Vollaard, “Hardware Dimensioning for Microservice-based Cyber-Physical Systems: A Profiling and Performance Prediction Method,” Vrije Universiteit Amsterdam, 2024.

ICT, Strategy & Policy

High Tech Campus 25
5656 AE Eindhoven
www.tno.nl

TNO innovation for life