Informatics

Technical Report

The MDM-CPPS Framework: GitOps-enabled Multi-Domain Modeling in Cyber-Physical Production Systems Engineering

Report No.: CDL-SQI 2024-01
Issued: 2024-01-22

Felix Rinker [1,2]
Kristof Meixner [1,2]
Diana Vysoká [1,2]
Stefan Biffl [1,2]

[1] Institute of Information Systems Engineering, TU Wien, Vienna, Austria
[2] CDL for Security & Quality Improvement in the Production System Lifecycle, TU Wien, Vienna, Austria

Abstract

Engineering Cyber-Physical Production Systems (CPPSs) requires the collaboration of engineers from different domains. Usually, engineers work in domain-specific work environments. The management of such domain-specific views and the artifact exchange between them is challenging because of diverging concepts. To improve the overall engineering process a holistic view of the engineering concepts and common model should be established. In this paper, we introduce a method for Multi-Domain Modeling for CPPS (MDM-CPPS) that allows engineers to define local concepts in a distinct view and negotiate a holistic view based on common concepts in a collaborative effort. The method then allows to independently change the properties in the domain-specific views and merge them back in a structured process based on DevOps and GitOps practices. To this end, we provide an architecture that incorporates a toolchain consisting of a Domain-specific Language (DSL) for modeling domain-specific concepts and common concepts by a Language Server Protocol (LSP) supported editor, and services to generate domain-specific views on the common model as well as delta comparison and merge capabilities automated by a continuous integration pipeline. Changes to the domain-specific model are semantically analyzed and the impact on other domains is calculated before the changes are seamlessly integrated into the common model. The change impact resolving process is managed using issues and their workflow in an issue tracker and a Git-based source code repository. We follow the Design Science methodology to address the challenges of the multi-domain modeling in CPPS, introduce and evaluate the MDM-CPPS method, and propose the MDM-CPPS architecture and toolchain. We evaluated our method and architecture with a feasibility study based on a use case from the industry. The proposed MDM-CPPS approach provides a) the means of common system understanding, b) a reduced effort of change coordination by using DevOps practices, such as GitLab continuous integration functionality, and c) the management of assets of the CPPS project using GitOps practices.

Contents

Contents **iii**

1 The MDM-CPPS Framework **1**

1.1 Introduction 1
1.2 Background & Related Work 2
1.3 Multi-Domain Modeling for CPPS 3
1.4 Feasibility Study on an Illustrative Use Case 9
1.5 Discussion 13
1.6 Conclusion & Future Work 14

A Annex: Acronyms **17**

B Annex: List of references **19**

Bibliography **21**

The MDM-CPPS Framework 1

1.1 Introduction

Cyber-Physical Production Systems (CPPSs), such as highly automated *car manufacturing plants*, have gained significant interest over the past decade. Such systems use the latest information and communication technology to interact autonomously with the physical assets to manufacture customized products from a product portfolio with modern production techniques [13, 9].

Designing and building the intended CPPSs involves engineers from multiple domains, such as *functional, mechanical, electrical*, and *automation* engineering (cf. use case *Position-and-Screw Robot Cell* [17]). Therefore, it requires accurate and reliable modeling of the domain-specific concepts to define common concepts to foster a shared understanding [7] for improving engineering quality and lowering effort.

However, due to diverging domain concepts, finding common concepts that support the collaborative engineering of assets is challenging. Based on the use case *Position-and-Screw Robot Cell* and the related work Traceable Multi-view Model Transformation (TMvMT) [17] and Multi-view Change Management (MvCM) [16] we identify challenges that hinder efficient Multi-Domain Modeling (MDM) for CPPSs. In the following, we discuss and categorize them according to the *DevOps4CPPS challenges* [10]:

Multi-disciplinary → C1. Domain-specific concepts and views result in an insufficient common system understanding. Usually, engineering disciplines work with domain-specific concepts, modeling languages, and tools. Improving the overall system understanding requires a process to get domain experts together to build an initial set of common concepts.

Agility → C2. High effort to coordinate changes and their impact across multiple domains. Heterogeneous engineering teams often start their work with incomplete information that is elaborated over time. Tracking changes in these domains and calculating the impact on other disciplines requires a multi-domain change coordination process.

Integration → C3. Scattered and heterogeneous domain knowledge hinders a holistic system view. Heterogeneous software landscapes and communication systems require complicated update processes. Tool suites in CPPS engineering often provide restricted data management support beyond their limited scope. A holistic view of the engineering concepts, model, and system should be established to improve the overall engineering process quality.

This paper introduces a method for multi-domain CPPS modeling that allows engineers to define local concepts in a distinct view and negotiate a holistic view of the common concepts in a collaborative effort. The method allows us to independently change the properties in the domain-specific model and merge them back in a structured process based on GitOps to the common model.

We design a toolchain with a Domain-specific Language (DSL) for modeling domain-specific concepts and common concepts using a Language Server Protocol (LSP) supported editor. We provide services to generate domain-specific views of the common model, delta comparison, and merge capabilities.

Changes to the common model are semantically analyzed, and the impact on other domains is calculated. The change impact resolving process is managed using issues and their workflow in an issue tracker and a Git-based source code repository.

We follow the Design Science methodology to address the challenges of improving the multi-domain modeling efficiency in CPPS engineering. We evaluate our method and toolchain in a feasibility study on a use case from the car manufacturing industry.

The remainder of this work is organized as follows: Section 1.2 discusses related work on system and knowledge modeling, multi-domain change management, and Dev- and GitOps for CPPS. In Section 1.3, we present the solution approach to organizing multi-domain engineering projects and conducting domain-specific change management in CPPS organizations. In Section 1.4, we describe the application of the method to an industry use case and discuss our results. Finally, we conclude our work and give an outlook on future work in Section 1.6.

1.2 Background & Related Work

This section summarizes relevant related work in the the fields of *Multi-domain Systems Modeling*, *Multi-domain Change Management*, *Knowledge Representation & Knowledge Graphs* and *DevOps and GitOps for CPPS.*

1.2.1 Multi-Domain Systems Modeling

In Cyber-Physical Production System (CPPS) engineering, domains' multi-disciplinary nature and technical divergence require the explicit specification of the system knowledge, dependencies, and relations between different system view [21]. System and domain knowledge can be efficiently modeled using the Product-Process-Resource (PPR) notation [18] for which the VDI 3682[1] provides a visual and formal representation. The PPR-DSL [12] provides a Domain-specific Language (DSL) that complements the notation.

Wortmann et al. conducted a systematic mapping study on state-of-the-art modeling languages for CPPSs [22]. They propose combining such model-based approaches with DevOps practices, e.g., integrating models analogously to code, as known from software engineering. We will implement the seamless model-based integration in our approach as described in the Multi-view Modeling Framework (MvMF) [17].

1.2.2 Multi-Domain Change Management

[16] proposed the Multi-view Change Management (MvCM) workflow for CPPS that enables engineers to implement changes to the CPPS while tracking the impact on other engineering domains. We proposed using *Pull Requests* to integrate changes from different domains, as described in [11]. Pull Requests are not an out-of-the-box part of the Git technology but state-of-the-art in Git-based source code repositories, such as GitHub, GitLab, or Bitbucket. The critical point here is to ensure that (other) domain engineers review the new implementation to ensure consistency and that the recent changes do not have a negative impact on the different parts of the CPPS. To support the engineers in conducting such a review, we will build on top of the related work to 1) implement the conceptual workflow on an actual use case and 2) implement semantic analysis functionality that identifies impacted assets of the CPPS, before the changes are integrated into the common model.

Xie and Ma [23] address a similar issue, focusing on complex engineering projects in collaborative domains, in which engineering change propagation is usually conducted manually without a systematic consistency examination. To address the issue, they propose an engineering constraint modeling method that supports the engineers in change evaluation results and conflict suggestions during engineering change implementation. It also helps identify the admissible value ranges of constraint parameters to accept design changes quickly or identify potential design conflicts based on a feature parameter association map that provides an evaluation context.

1.2.3 Knowledge Representation & Knowledge Graphs

Arenas, Gutierrez, and Sequeda [2] define Knowledge Graphs as *"a software object (artifact) that represents (codifies), integrates and produces knowledge."* To implement knowledge graphs, graph databases such as *Neo4j, ArangoDB* or *GraphDB* can be used. We will use Neo4j[1] in our system design. Hildebrandt et al. have proposed an approach to build ontologies for CPPS that could later be stored in a graph database as a knowledge graph[8].

In the CPPS engineering, domain-specific representations of the production assets need to be integrated into a holistic view. Grangel et al. [6] propose a knowledge-graph-based approach for semantically integrating manufacturing data. The proposed Bosch Industry 4.0 Knowledge Graph uses Common Concepts (CCs) to describe relations and semantically integrate data from different data sources.

Rinker et al. proposed a Multi-Domain Engineering Graph (MDEG) to represent production knowledge as a graph [16]. MDEG uses *Common Concept Glossary* as an approach to establish a holistic cross-domain understanding of the CPPS assets and their characteristics. We will use the MDEG model to represent the knowledge graph to describe the industrial use case.

1.2.4 DevOps & GitOps for CPPS

DevOps practices have recently gained interest in the Model-Driven Engineering (MDE) and CPPS community. Süß et al. describe that MDE tooling is not designed to participate in DevOps pipelines, which makes the adoption of MDE in the industry more difficult [19]. Their work describes how they apply DevOps practices to enable MDE adoption.

GitOps is a software development and deployment methodology that applies Git-based project repositories as single source of truth for infrastructure and application code. This methodology aims to increase efficiency, reliability, and traceability by applying version control features, such as branching, merging, and continuous integration/continuous deployment (CI/CD) pipelines[4]. We base our architecture on the GitOps methodology, using branching, merging, and continuous integration.

Halilaj et al.[7] proposed a method to collaboratively define a cross-domain vocabulary and consensus among experts from different domains using a Git-based workflow. The complexity of the process to achieve this goal is increased with the number of people involved and the variety of the systems to be integrated. Defining such cross-domain vocabulary is part of our *project setup* phase, in which the engineers create domain *Concept Glossaries* and *Common Concept Glossary.*

1.3 Multi-Domain Modeling for CPPS

In this section, we propose a) the Multi-Domain Modeling for CPPS (MDM-CPPS) method to combine domain-specific concepts and models into an integrated engineering model and b) the MDM-CPPS architecture to automate the MDM-CPPS method in agile multi-domain modeling environments.

1.3.1 MDM-CPPS Method

The Multi-Domain Modeling (MDM) method for Cyber-Physical Production System (CPPS) engineering consists of two phases, the *project setup phase* and the *multi-domain change management* phase, which are explained in the following sections. The *project setup* phase allows CPPS engineers to define their local concepts in a distinct view and negotiate a holistic view of their common concepts collaboratively. The *multi-domain change management phase* defines a coordinated process to integrate changes from the domain-specific view into the common view.

[1]Neo4j: `https://neo4j.com`

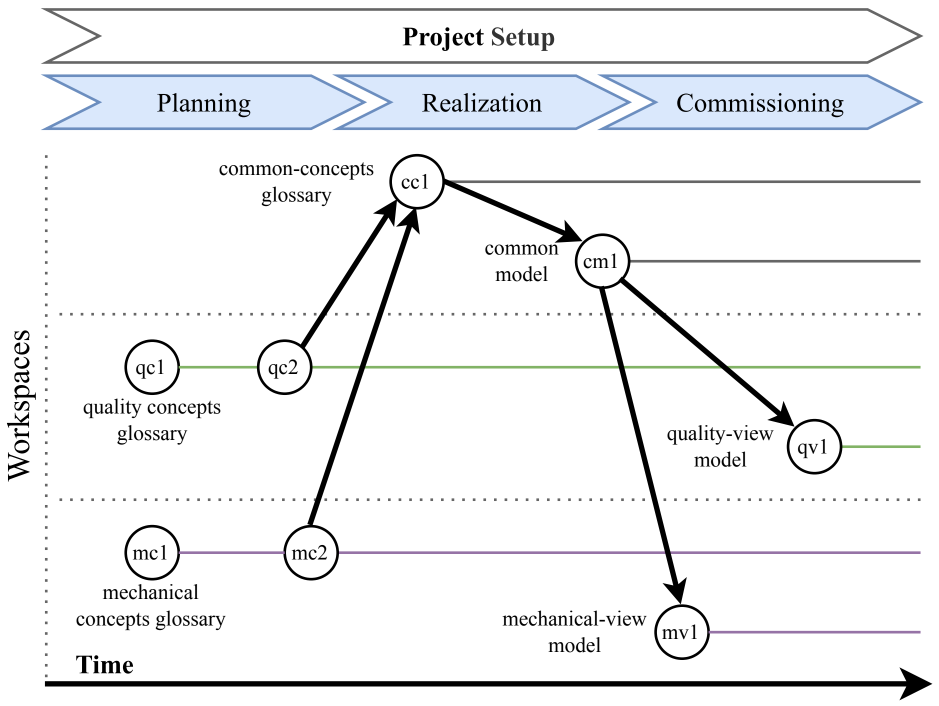

Figure 1.1: Project Setup Phase of the MDM-CPPS Method.

1.3.2 Multi-Domain Modeling for CPPS – Project Setup

The *project setup* phase is divided into the three activities *planning*, *realization*, and *commissioning*. Figure 1.1 shows this phase and the activities, which we detail in the following.

Planning. In the *planning* activity, the engineers set up *local workspaces* for each domain within the project. These domain-specific workspaces serve as dedicated environments for concept development for the domain-specific teams. Within these spaces, they define and work in distinct glossaries on *local concepts* tailored to the unique requirements of their domains. For instance, Fig. 1.1 shows *mechanical* and *quality concept glossaries*, labels *mc1* in the mechanical and *qc1* in the quality workspace, for the CPPS engineering project. An example would be the mechanical definition of the attributes of a screwdriver. These defined concepts can also evolve throughout domain-specific discussions (Fig. 1.1 labels *mc2* and *qc2*).

Furthermore, those workspaces have the flexibility to accommodate additional domain-specific models.

Realization. In the *realization* activity, a lead engineer first sets up a *common workspace*. This way, the engineer employs a shared hub to ensure effective collaboration for an integrated engineering team, where each domain involved in the engineering of the CPPS must be represented. These concepts represent a taxonomy that can be well modeled in a knowledge graph. Then, in a negotiation process, within the integrated team, the engineers agree on and describe *common concepts* for the CPPS in a *Common Concept Glossary* (Fig. 1.1 label *cc1* in the common concepts workspace) from their local concepts. For example, they agree on a taxonomy for screwdrivers, differentiating between electric and pneumatic screwdrivers and their attributes relevant to the particular domains. Additionally, relations can be defined on the concept, common concept, and asset level. A relation defines a dependency between attributes of two dependent objects. Afterward, they map the previously defined local concepts from their glossaries to the common concepts in the *Common Concept Glossary* in a Git-based workflow [7]. This includes, e.g., transformations of values between the different domains, like the electrical power calculation to actual revelations.

In a second activity, the lead engineer, supported by the integrated team, creates a common model of the CPPS (label *cm1* in the common concepts workspace), e.g., the functional view, that is iteratively refined. This workspace now contains essential assets, including common concepts and a unified model based on a Single Underlying Model (SUM) [20]. This agreed-on SUM forms the basis for the engineering project, ensuring consistency and coherence across domains.

Commissioning. In the *commissioning* activity, the domain-specific models (Fig. 1.1 labels *qv1* and *mv1* in the local workspaces) are derived based on the local concepts, containing only domain-specific assets. This view allows teams to focus on domain-specific concerns while aligning with the overarching project structure.

Changing attributes of the local concepts and merging them back to the common model requires a coordinated change management process described in the next section.

1.3.3 Multi-Domain Modeling for CPPS – Engineering and Change Management

Figure 1.2 shows the *engineering and change management phase*, with the *engineering*, *merge request & review*, and *merge* activities, which are explained in the following.

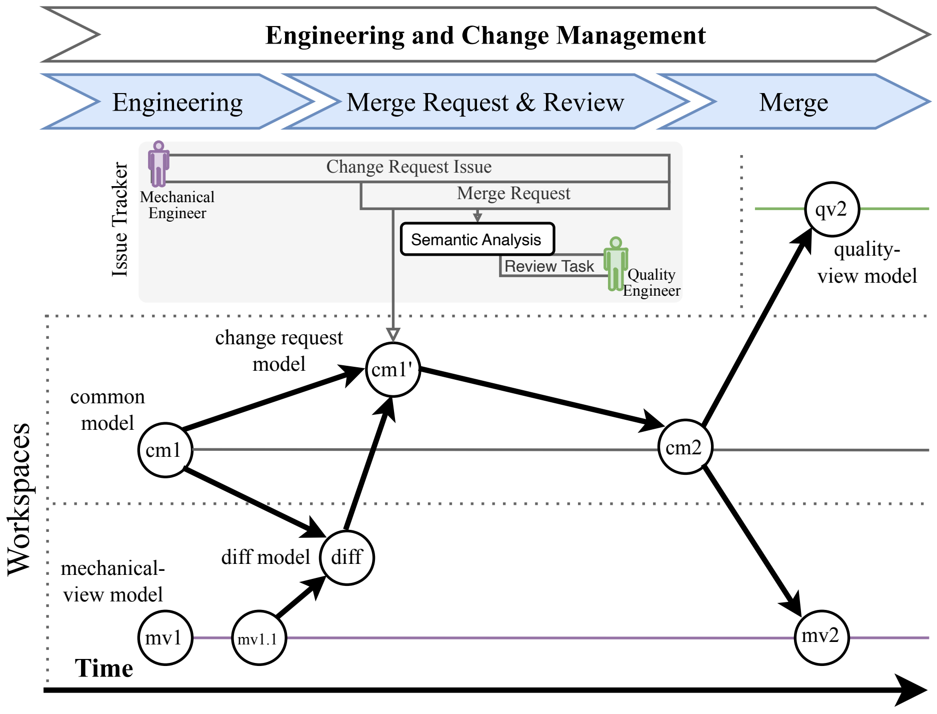

Figure 1.2: Engineering and Change Management Phase of the MDM-CPPS Method.

Engineering. Throughout the *engineering* activity, initiated by a *change request* specified in an issue in an issue tracker, the engineers of the respective domain work in their workspace adapting the domain-specific model. Each of these changes creates a new version of the local model in their particular workspace. For instance, the mechanical engineers change the attributes of a screwdriver (Fig. 1.2 label *mv1* to *mv1.1* in the mechanical workspace). This step is relatively untouched by the activities in the other domain workspaces.

Merge Request & Review. The *merge request & review* activity comprises several steps. For the following explanation, we differentiate between the initiating and affected domains.

The engineers of the initiating domain aim at merging their local changes into the *common model* (Fig. 1.2 label *cm1*), which comes from the *project setup phase*. For the MDM-CPPS method, we employ an extended concept of the pull request (later referenced as merge request due to its naming in GitLab) of the Git Workflow for Change Management from related work [15].

First, the method requires the engineers to create a *merge request* in the issue tracker. Second, a *diff model* (label *diff*) is calculated from the local model of the initiating domain, and the latest version of the *common model* (labels *mv1.1* and *cm1*), to examine the changes. This way, the domain engineers can check their changes with the *common model*.

Then, for each of the domains involved in the project, a *semantic change impact analysis* investigates if the change affects their local model. Therefore, a *change request model* (Fig. 1.2 label *cm1'*) must be calculated regarding the *common model* and the *diff model* (Fig. 1.2 labels *cm1* and *diff*) that is used to investigate whether concepts of the local models are affected. This means that for each local concept, the common concepts need to be identified, and from these, the local concepts linked to the common concepts need to be found. Then, for each of the involved domains, a *change request model*is created based on the semantic change impact analysis. For the affected domains, *change review tasks* are created and assigned to the engineers of the affected domains.

Finally, the consent and acceptance of that change are discussed and decided upon within the *change review* and *merge request* issues. If the affected domains accept the change, it will be merged into the *common model* in the merge activity. However, if the change is declined, the engineers of the initiating domain need to address rework tasks, considering the comments and decisions made during the review process.

Merge. In the merge activity, the *change request model* (Fig. 1.2 label *cm1'*) are merged into the *common model* (label *cm1*) creating a new version (label *cm2*). In the second step, the changed values in the *common model* are propagated to the initiating domain model and the affected domain models (labels *mv2* and *qv2*).

1.3.4 Multi-Domain Modeling for CPPS – Semantic Analysis

This subsection specifies how the semantic analysis is carried out in the context of the *Merge Request & Review* activities.

Change identification. To identify changes to the assets, e.g., a screwdriver, we compare the contents of the *common model* with the contents of the given domain-specific model. We define a set of assets in the *common model* as A_{cm} and a set of assets in local domain views as A_d, where a_d is a member of A_d and a_{cm} is a member of set A_{cm}. First, we iterate through the elements of A_d and find a corresponding element in A_{cm} by its qualified name q. Once we find both elements, we compare the list of attribute values.

If the value of an attribute in a_d defined in a domain-specific model does not correspond to the value of an attribute in a_{cm}, then add the old value, new value, q of the attribute, change type *CHANGE* and qualified name q of a_d to the change set. Should the attribute not be present in the domain-specific or common model, this situation is considered faulty since we do not allow deletion or adding attribute values to assets.

Semantic analysis. To conduct the semantic analysis, we first build a knowledge graph $G = (V, E)$, where concepts, common concepts, and assets belong to the set of vertices V, and dependency relations form the graph's edges E.

We define $C_{a,v}$ as a set of changes, from the *change identification* step, that includes a pair p of asset a and an attribute value v, notated as $p = (a, v)$. For each changed attribute value v and its parent asset a, we traverse the knowledge graph and find the relation R, in which v is either the start or end node. We then return a resulting set of impacted assets and their attribute values $I_{a,v}$.

Table 1.1: Formal notations

Symbol	Description
R	Relation
cc_{from}	Common Concept as a starting node of a relation
cc_{to}	Common Concept as an ending node of a relation
CC	set of Common Concepts
ACC_{from}	Set of assets that represent a Common Concept defined as a starting node of a relation
ACC_{to}	Set of assets that represent a Common Concept defined as an ending node of a relation
c_{from}	Concept as a starting node of a relation
c_{to}	Concept as an ending node of a relation

We only traverse the knowledge graph to *depth=1*, and we are only interested in immediately impacted assets and their attribute values.

Semantic Relation handling. In *semantic analysis*, we described how we use relations between assets to identify impact. In *realization* part of the *project setup* phase, we propose a definition of relations on the concept and common concept level.

First, to semantically differentiate these relations in the *engineering and change management* phase, we define two types of semantic links: *Common Model Semantic Links,* that are allowed in the common model; and *(Common) Concepts Semantic Links,* that are allowed in common concept and concept glossaries. To interpret (common) concepts dependencies on the object level of the common model, we transform the *(Common) Concept Semantic Links* to *Common Model Semantic Link* based on the following rules using notations shown in Table 1.1:

Link between common concepts (and attributes of the concepts they inhabit): given a relation R between Common Concept cc_{from} and Common Concept cc_{to}. For both common concepts, we find the set of assets ACC_{from} and ACC_{to} that represent the starting and ending common concepts defined in the relation R. We build a cartesian product of ACC_{from} and ACC_{to} and for each pair p = (acc_{from}, acc_{to}), where acc_{from} owns the attribute defined as starting attribute of R and acc_{to} owns the attribute defined as ending attribute of R, we finally create *Common Model Semantic Links* between the elements of the eligible pairs of the cartesian product.

Link between concepts (and attributes they own): We proceed similarly as in the case of links between common concepts, but first, we find the set of common concepts CC_{from} that inhabit the c_{from} and set of common concepts CC_{to} that inhabit c_{to} in R. Then, we create a cartesian product of CC_{from} x CC_{to}. For each resulting pair p = (cc_{from}, cc_{to}) of the cartesian product, we find a set of assets that represent a cc_{from} and cc_{to}. Finally, we proceed as in the case of links between common concepts.

1.3.5 MDM-CPPS Architecture

To automate the MDM-CPPS method, we propose the MDM-CPPS architecture design that is depicted in Fig. 1.3 and contains three main components: a) the MDM-CPPS Framework b) the IDE Ecosystem c) the GitLab Ecosystem.

The system design is an advancement of the Multi-view Modeling Framework (MvMF) [17] that is inspired by the Eclipse Modeling Framework (EMF) [5]. However, since EMF is strong coupled with Eclipse[2] a setup in custom software solutions is hindered [3]. In the following, the components of the envisioned system are described in detail:

MDM-CPPS Framework. The framework provides functionality for describing multi-domain CPPS projects using the *MDM-CPPS Domain-specific Language (DSL)* and graph-based analyses and change management using the *MDM-CPPS Multi-Domain Engineering Graph (MDEG) API.*

[2]Eclipse: `https://www.eclipse.org`

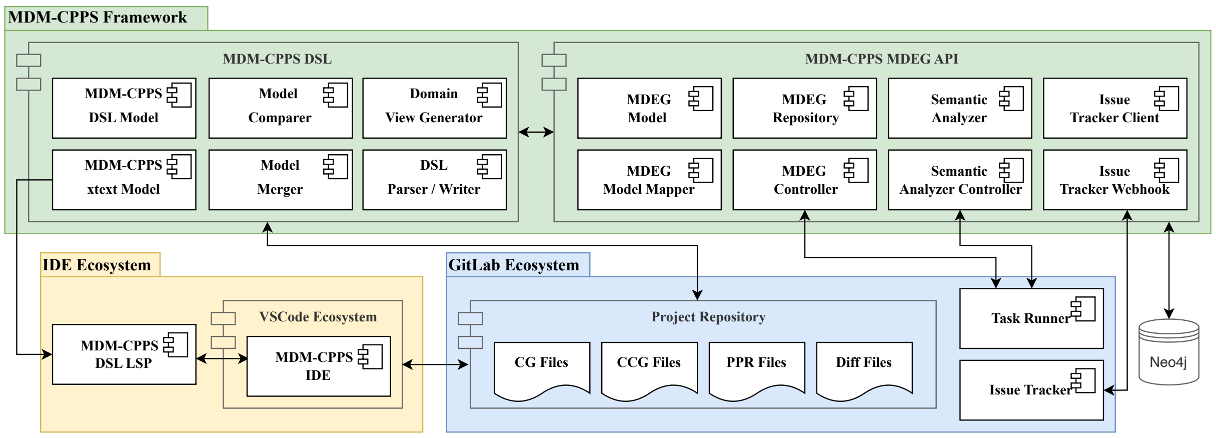

Figure 1.3: System Design of the MDM-CPPS Change Analysis and Management System.

MDM-CPPS DSL. To enable a formal description of a CPPS, we propose the *MDM-CPPS DSL* component. Each of the subcomponents is described below.

MDM xtext Model. The xtext grammar definition is an extension of the Product-Process-Resource (PPR) DSL [12], that formally defines the language for domain-specific concept and common concept modeling, PPR asset definition based on common concepts, relation modeling between the aforementioned objects.

MDM-CPPS DSL Model. The DSL model extends the PPR DSL [12] data model with domain-specific concept and common concept modeling entities.

Model Comparer. The comparer clones the *Project Repository*, containing the common and domain-specific workspaces, to the local file system. To identify changes made in a domain-specific model compared to the common model, the *DSL Parser* parses the domain-specific view file and the common model in two instances of a *MDM-CPPS DSL Model.* Finally, the comparer compares the domain-specific model against the common model and returns a collection of changes as a *Diff file.*

Model Merger. The merger applies the changes from the *Diff file* to the common model.

DSL Parser / Writer. This subcomponent provides functionality to read and write *MDM-CPPS DSL* files. The parser is realized by Antlr Parser Generator[3] that is based on *MDM xtext Model.* The writer is a custom solution implemented by us that prints the *MDM-CPPS DSL Model* to domain-specific files in the domain workspace.

Domain View Generator. The generator reads the concept, common concept, and common model files using *DSL Parser* and generates domain-specific view files based on the *MDM-CPPS DSL Model.*

MDM-CPPS MDEG API. The MDM-CPPS MDEG API component enables further semantic analysis of the data described in the *MDM-CPPS DSL* files. It exposes APIs to build a knowledge graph based on DSL data and APIs to interact with it. We describe the API's components below.

MDEG Model. This model contains entities to build a multi-domain engineering graph in Neo4j with appropriate relation references and database-related annotations.

MDEG Model Mapper. To represent the data described in *MDM-CPPS DSL* in a knowledge graph, we map the DSL domain model *(MDM-CPPS DSL Model)* to the graph domain model (MDEG Model).

MDEG Controller. This controller enables the creation of an MDEG knowledge graph based on the MDM-CPPS DSL files.

MDEG Repository. The repository is the data access layer that establishes a connection with the graph database and runs queries on it.

Semantic Analyzer. The analyzer calls the *Model Comparer* of the component MDM-CPPS DSL to identify the changes in domain-specific models compared to the common model. For each element of the identified change collection, the analyzer looks for impacted assets and their attributes in the knowledge

[3]Antlr Parser Generator: `https://www.antlr.org`

graph. Finally, it returns the change and its corresponding change impact information. Alternatively, it can use the *Issue Tracker Client* to create review tasks for the impacted elements.

Semantic Analyzer Controller. This controller enables semantic analysis based on the MDM-CPPS DSL files. The controller calls the *Semantic Analyzer*.

Issue Tracker Client. The client is called by *Semantic Analyzer* to create review tasks in the issue tracker as proposed by [15]. In our solution, we use GitLab[4] as an open-source issue tracker, as it provides REST API to interact with it. Once the review tasks are created, the client links them to the initial change request task to ensure change traceability. The freshly created review tasks are also referenced in the merge request created manually by an engineer for the initial change request.

Issue Tracker Webhook. To propagate the changes of the review tasks to our backend so that the *Issue Tracker Client* can track the status changes in the corresponding merge request, we expose an *issue webhook* that is triggered by the issue events in GitLab. To propagate the merge request changes to our backend, once the merge request is merged, we expose a *merge request webhook*, which GitLab triggers on merge request events. Then, the *Issue Tracker Client* calls *Model Merger* to merge model changes from domain-specific views to the common model and pushes it to the *Project Repository*.

IDE Ecosystem. The Integrated Development Environment (IDE) ecosystem is an environment for source-code editors, such as the *VSCode Ecosystem*[5], and language servers that implement the Language Server Protocol (LSP)[6]. The *MDM-CPPS xtext Model* defines the syntax and grammar for the MDM-CPPS DSL. The *MDM-CPPS DSL LSP* implements the LSP for this model and provides support for syntax validation and completion. The *VSCode Ecosystem* provides a plugin infrastructure for extending its functionality. The *MDM-CPPS IDE* plugin provides the integration of the *MDM-CPPS DSL* into the *VSCode Ecosystem* as source-code editor.

GitLab Ecosystem. To implement our solution, we need a tool ecosystem that supports Git-based repositories and issue tracking, in our case, GitLab. It is also necessary that the ecosystem provides APIs to interact with it. GitLab enables us to create merge requests that trigger the CI/CD pipeline with various tasks running on the source files. These pipelines are executed by the *Task Runner*.

Project Repository. The project files are located in the Git-based project repository. Each domain workspace contains *Concept Glossary* files with concepts and domain-specific view PPR files. The common workspace contains *Common Concept Glossary* file with common concepts and a common model PPR file.

Issue Tracker. The change requests, requirements, and other tasks are defined as issues in the GitLab *Issue Tracker*.

Task Runner. Once a merge request is created to merge domain-specific model changes to the common model, the GitLab CI/CD Pipeline is run to create a fresh knowledge graph using *MDEG Controller* and then calls *Semantic Analyzer*, which analyzes the impact of the changes in domain-specific view files and finally creates review tasks in the Issue Tracker.

1.4 Feasibility Study on an Illustrative Use Case

For the evaluation of the Multi-Domain Modeling for CPPS (MDM-CPPS) method, we adapted the *Fasten Screw & Measure* process, shown in Fig. 1.4, of the *Position-and-Screw Robot Cell* use case [17]. We assume a car body with an inserted dashboard in this use case. A screwdriver on a robot arm needs to fasten the screws and measure the result for accuracy and torque. The result of the process is a dashboard fixed to the car body.

The *Electric Screwdriver* and *Robot* are examples of common concepts instantiated as resources, with attributes and dependency links to other resources or processes. We will showcase the *MDM-CPPS*

[4]GitLab: `https://gitlab.com`
[5]VSCode: `https://code.visualstudio.com`
[6]LSP Protocol: `https://microsoft.github.io/language-server-protocol`

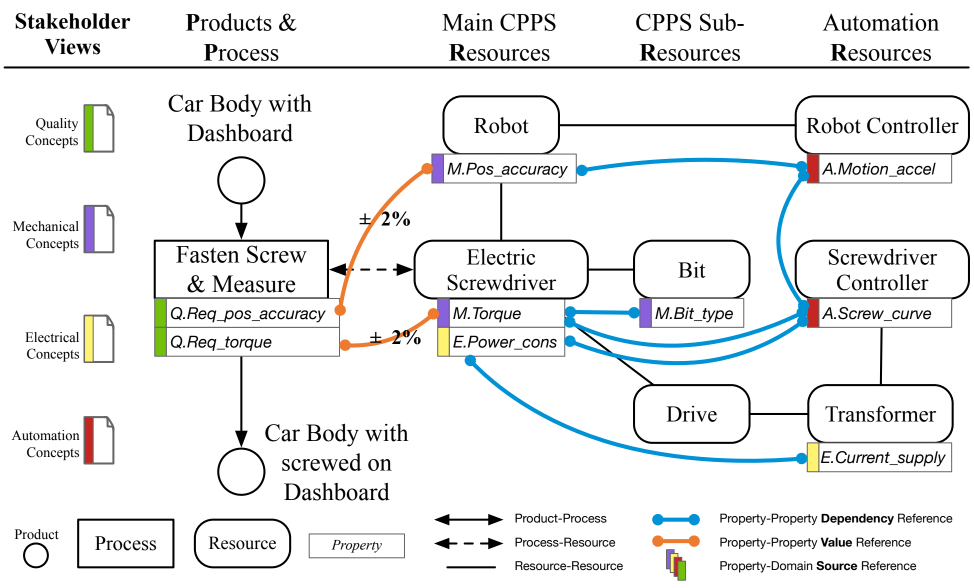

Figure 1.4: Multi-Domain Engineering Graph of the process *Fasten Screw & Measure*: with dependency links, based on [17]

method's functionality during the *change request & review* phase (cf. Sec. 1.3.3) with two types of links: Semantic relations, e.g., *M.Torque* references to the values *M.Bit_type* and *A.Screw_curve* for change impact analysis. Value references between attribute values, e.g., *Q.Req_torque* and *M.Torque* value equality with tolerance range.

In the following, we describe the *project setup* and *engineering and change management* phases utilizing the process *Fasten Screw & Measure* (cf. Fig. 1.4).

1.4.1 Project Setup Phase

This Section evaluates the *project setup phase* for the use case. First, domain-specific concepts are defined and consolidated into common concepts. Then, the common model is created based on the common concepts, and cross-domain dependencies are defined between assets and their attributes.

Concept definition. (cf. Section 1.3.1 - planning). Domain-specific concepts are defined in *Concept Glossary* files located in the domain workspaces with a file extension *.cg*. Concepts have an `ID` and two fields; name and `attributes`. Attributes are also defined in the domain *Concept Glossary* having a `name`, `defaultValue`, `type`, and `unit` field. Listing 1 shows a *Concept Glossary* definition of the mechanical domain. It starts with the glossary id definition *MechanicalConcepts*. Then, the concept *c_m_electric_screwdriver* with its name *Electric Screwdriver* and the attribute *torque* is defined.

Common Concept definition. (cf. Section 1.3.1 - realization). Common Concepts (CCs) are defined in a *Common Concepts Glossary* file in the *common workspace* with the file extension *.ccg*. The *Common Concepts Glossary* is specified with a unique `ID`, a `name`, and `version` field. A CC has a unique `ID` and the two fields `name` and `inhabits`, where the latter is a list of concepts inhabited by a CC. Listing 2 shows the CC with id *cc_electric_screwdriver* that inhabits the mechanical specific concept *c_m_electric_screwdriver* and the electrical specific concept *c_e_electric_screwdriver*.

Common Model modeling. (cf. Section 1.3.1 - realization). Product-Process-Resource (PPR) assets are modeled in a *Common Model* file in the *common workspace* with the file extension *.ppr*. The *Common Model* has also an identification header with a unique `ID`, a `name`, and a `version` field. *Product*, *Process*

```
ID MechanicalConcepts {
  name: "Mechanical Domain Concepts Glossary"
  version: 1.0.0
}
Attribute torque {
  name: "torque"
  defaultValue: 0.0
  type: "Number"
  unit: "Nm"
}
Concept c_m_electric_screwdriver {
  name: "Electric Screwdriver"
  attributes: torque
}
```

Listing 1: Illustrative *Concept* definition *Electric Screwdriver* with attribute *torque* for the mechanical domain.

```
ID CommonConceptGlossary {
  name: "Common Concept Glossary"
  version: 1.0.0
}
CommonConcept cc_electric_screwdriver {
  name: "Electric Screwdriver"
  inhabits:
    MechanicalConcepts.c_m_electric_screwdriver,
    ElectricalConcepts.c_e_electric_screwdriver
}
```

Listing 2: Illustrative negotiated *Common Concept* definition *Electric Screwdriver*, summarizing related *Concepts* from the mechanical and electrical domain.

or *Resource* assets starts with an unique `ID` and have at least two fields `name` and field `represents`, where the latter refers to a CC.

Listing 3 shows the defined resource with id *electric_screwdriver* that represents the CC *cc_electric_screwdriver* and thus also specifies the domain-specific attributes mechanical *torque* and electrical *power_consumption*.

```
ID PositionScrewDashboard_Model {
  name: "Model Fasten Screw and Measure Use Case"
  version: 1.1.0
}
Resource electric_screwdriver {
  name: "Electric Screwdriver"
  represents: CommonConceptGlossary.cc_electric_screwdriver
  children: bit
  parents: robot
  requires: drive
  ElectricalConcepts.power_consumption: 0.0
  MechanicalConcepts.torque: 0.0
}
```

Listing 3: PPR *Common Model* definition *Electric Screwdriver* representing the *Common Concept Electric Screwdriver* with attributes *power_consumption* and *torque*.

Relation definition. Each of the *Concept*, *CC* and *Common Model* files can contain relations. A Relation is defined with a unique `ID` followed by the field `from`, which indicates the starting node of the relation, which is a combination of an asset and its attribute. Next, the field `to` describes the target and consists of a combination of an asset and its attribute specification, indicating the end node of the relation. The `definition` field indicates the relation type between the attribute values.

Listing 4 shows the relation with id *relation6_screwing_tension*. The `from` field links to the *common model* asset *electric_screwdriver* mechanical attribute *torque*. The `to` field links to the *common model* asset *screwdriver_controller* automation attribute *screw_curve*. The example shows that the two attributes can be linked depending on each other using *dependency* `definition`. Alternatively, we could use *propagate* as `definition` to indicate that the value of one attribute should be propagated to the other attribute based on some value mapping or transformations of the relation. This functionality is currently only conceptually defined, based on the reactive links concept [14], and will be realized in future work.

```
Relation relation6_screwing_tension {
    from: electric_screwdriver -> MechanicalConcepts.torque
    to: screwdriver_controller -> AutomationConcepts.screw_curve
    definition: "dependency"
}
```

Listing 4: PPR *Relation* definition for the `dependency` between the quality and the mechanical attribute *torque*.

Domain-specific View generation. To enable domain engineers to implement their change requests in a domain-specific context, *domain-specific view* files are generated. Domain-specific view files start with a comment generated based on the *common model*. The `ID` is the unique identifier of this model and also includes a *name* and a *version* field. All domain-specific assets and relations are included.

PPR assets are inherited into a local file with its full qualified name (e.g. *PositionScrewDashboard_Model.electric_screwdriver* cf. Listing 5) including the field `represents` and a list of domain-relevant attributes. The fields `children, parent, excludes, implements, requires` are optional. We ignore the `name` and abstract flag as we do not allow change of these fields in domain-specific views. Additionally, the value of the attributes is inherited from the common model. If the attribute value is missing in the model, we inherit the attribute's default value from the domain concept definition.

Listing 5 shows the generated domain-specific view file for the mechanical domain. A mechanical engineer changes the domain-specific attribute mechanical *torque* to the value *10.0*, in the *cc_electric_screwdriver*.

```
// Generated from 'FastenScrewDashboard_Model' with version 1.1.0
 ID MechanicalView {
        name: "Generated Mechanical View based on 'FastenScrewDashboard_Model'"
        version: 0.1.0
}
Resource FastenScrewDashboard_Model.electric_screwdriver {
        name: "Electric Screwdriver"
        represents: CommonConceptGlossary.cc_electric_screwdriver
        children: FastenScrewDashboard_Model.bit
        parents: FastenScrewDashboard_Model.robot
        MechanicalConcepts.torque: 10.0
}
```

Listing 5: Mechanical *View Model* with `Resource` *Electric Screwdriver* with mechanical attribute *torque*, as result of the domain-specific view generation.

Additionally, if relations are defined in the common model, we use the following logic to inherit them to domain-specific view files: given a relation R with two assets A_{from} and A_{to} and their attributes p_1 and p_2. If both attributes are defined in the domain concepts file, then inherit this relation to the domain-specific view file. If at least one of the attributes is not defined in the domain concept file, the relation is not inherited.

1.4.2 Engineering & Change Management Phase

To enable the *Engineering & Change Management Phase*, we utilize the Section 1.4.1 results. In the *engineering phase*, attribute values are changed in domain-specific view models (e.g. using the *MDM-CPPS IDE* and based on change request documented in the *Gitlab Issue Tracker*). For example, in the mechanical view, the *torque* value of the *electric screwdriver* is changed from *0.0* to *10.0* (cf. List. 5). When the task in the change request is fulfilled, the change can be propagated to the common model.

Domain-to-Common Model comparison. To propagate a change to the common model using *merge request*, the semantic differences between the domain view model and the common model are calculated using the *Model Comparer*. The resulting diff model is stored as *json* object in a temporal diff file in the domain workspace. A *diff object* is, for instance, an *attribute change* indicated by the `changeType` field and the related attribute with the *attribute* field. The parent element is referenced with the `pprAsset` field. The `valueOld` and `valueNew` fields contain the new and old values.

```
{"diffs": [{
"pprAsset": "FastenScrewDashboard_Model.electric_screwdriver",
"valueOld": 0.0,
"changeType": "CHANGE",
"attribute": "MechanicalConcepts.torque",
"valueNew": 10.0
}]
```

Listing 6: Difference Model with an attribute value change of *torque* in the mechanical domain.

Listing 6 shows the diff model of the *torque* value change. The `pprAsset` *FastenScrewDashboard_Model .electric_screwdriver* is the connecting reference between the domain view and the common model. `oldValue` and `newValue` showing the value change from *0.0* to *10.0* for the `attribute` *MechanicalConcepts.torque*.

Knowledge Graph generation. To enable the semantic analysis of the change impact on other domains, the common model is mapped to a Multi-Domain Engineering Graph (MDEG) using the *MDEG Model Mapper* and instantiated in a graph database using Neo4j.

Figure 1.5 shows the knowledge graph instance of the resource *Electric Screwdriver* (yellow) and *Bit* (yellow), the underlying common concepts (purple), the related mechanical concepts (red) with the attributes torque (left-blue) and bit type (right-blue). The value of an attribute is an instance (brown) and the connection between the attribute and the resource. Figure 1.5 also shows the CCG_DEPENDS_ON edge between torque and bit type. These blue nodes are attributes defined in the Concept Glossary, and the glossary contains a relation between these two attributes. To transform this edge to the asset level, we create a new edge between the brown value instance of the blue attribute nodes and call it PPR_DEPENDS_ON.

1.5 Discussion

The Multi-Domain Modeling (MDM) method for Cyber-Physical Production System (CPPS) introduced in this paper offers three pillars for cross-domain engineering of CPPS for agile production systems engineering. Specifically, we investigated how to automate the change management process of engineering changes that span through several domains. The evaluation was conducted on an automotive industry use case in the scope of a feasibility study. Based on model examples and comprehensive screencasts of selected parts of the prototype system, we demonstrated that the newly proposed method is feasible and efficiently applicable to the selected use case.

The first pillar, the *project setup*, enables the CPPS specification using *MDM-CPPS DSL*, and the files are structured in a Git-based *Project Repository*, that we manage using GitLab. GitLab is the central

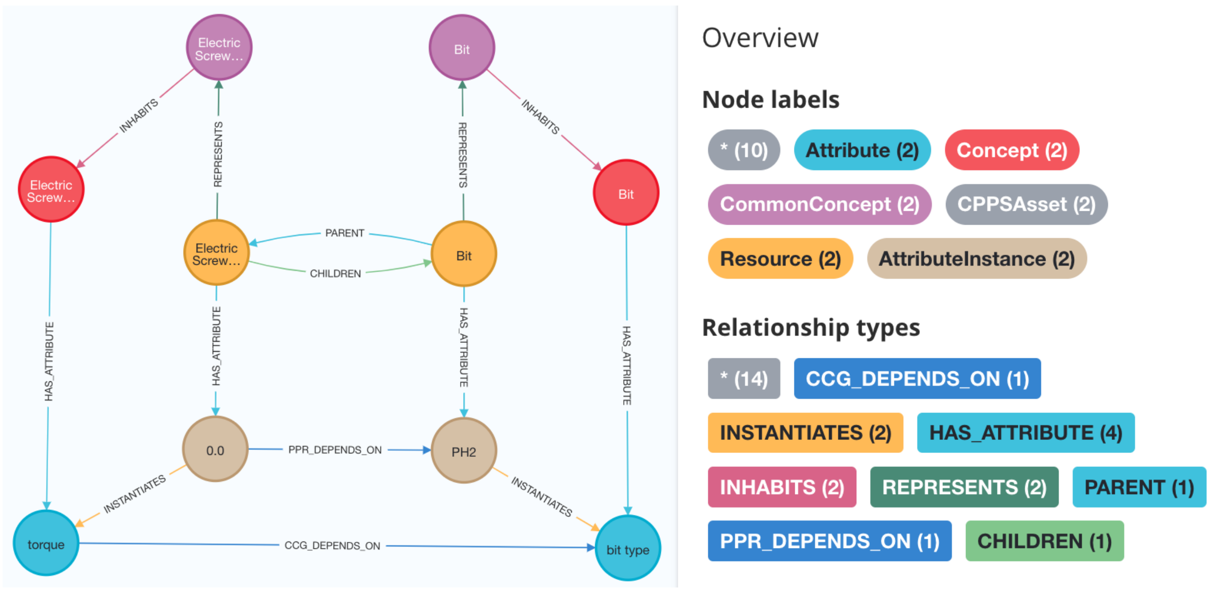

Figure 1.5: Multi-Domain Engineering Graph in Neo4j showing the resource *Electric Screwdriver* and its child resource *Bit*, including the relations on concept (CCG_DEPENDS_ON) and asset (PPR_DEPENDS_ON) level.

enabler in our proposal to apply GitOps practices. Using the Git branches for each dedicated change request, on which engineers from multiple domains collaborate and resolve review tasks, enables traceability and historicization of the MDM-CPPS files.

The second pillar of our method, *engineering and change management*, enables automation of the engineering change management and information-system-supported change review process based on merge requests, applying Git Workflow. Application of GitLab's merge requests enables us to integrate the Traceable Multi-view Modeling Transformations, as we run a custom model integration pipeline that analyzes the impact of fresh changes and integration of the domain-specific changes to the common model once the implementation is ready to be merged. This fosters the idea of GitOps in CPPS by using Git-based repositories always to have the single truth source of the CPPS configuration.

This pillar integrates semantic analysis of the engineering changes, encompassing the proposed method's third pillar.

The method addresses the identified challenges from Section 1.1, as it provides common system understanding in *(Common) Concept Glossaries* and common model (C1). The effort to coordinate changes and their impact is reduced by automation via the *Task Runner* and its pipeline (C2). The *Project Repository* with the domain workspaces provides a holistic system view and single-point data management (C3).

The traditional document-based approach is suitable for minor changes and non-complex engineering systems. However, large engineering projects in agile CPPS engineering require fast and precise implementation and consequential analysis of changes to minimize negative impact.

Therefore, we expect the MDM-CPPS method to require the necessary one-time effort for the project setup, including defining domain concepts, common concepts, and the common model. Then, provisioning of the information system that facilitates and automates the MDM-CPPS method is necessary. Once these pre-requirements are finalized, we expect the method to reduce the coordination and change management effort.

1.6 Conclusion & Future Work

Complex engineering environments, such as car manufacturing plants, require a stakeholder's coordination of engineering assets from multiple domains to keep their products competitive in a changing market. To implement a change to such a complex engineering system, it is essential that experts from various domains can work simultaneously on their changes without negatively impacting other domains while keeping the system consistent in each domain-specific model.

The novelty of this paper is a method for modeling CPPS from multi-domain perspectives supported by: a) a realization of the Multi-view Change Management (MvCM) Workflow[15] using a Git-based source code repository, in our case GitLab; b) a Visual Studio Code extension for multi-domain system modeling environment; c) an extension of the Traceable Multi-view Modeling Transformation Framework [17] to a MDM-CPPS framework providing a Domain-specific Language (DSL) for modeling the CPPSs and services for change management and impact analysis of a change on a knowledge graph. This functionality is driven by an issue tracker system and automated by an open source CI/CD environment, in our case GitLab, to provide an end-to-end toolchain for systems engineering. As described in Section 1.5, we addressed the challenges from Section 1.1 and provided means for engineers in multidisciplinary environments to coordinate multi-domain modeling activities efficiently.

Future Work. As a next step, we plan to investigate cases where the engineering change impacts other domains that cannot be reworked to meet the desired criteria in different domains. A promising extension to our solution can be reactive links [14] incorporated into the relations in our knowledge graph. Additionally, we plan to integrate risk-related aspects, such as Failure Mode and Effects Analyses (FMEAs), to the MDEG to enable validation of the attribute value based on specified ranges in the risk analysis to minimize production risks.

Acknowledgments

The financial support by the Christian Doppler Research Association, the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

Annex: Acronyms A

CC	Common Concept. 3, 10, 11
CPPS	Cyber-Physical Production System. ii, 1–5, 7, 8, 13–15
DSL	Domain-specific Language. ii, 1, 2, 7–9, 15
EMF	Eclipse Modeling Framework. 7
FMEA	Failure Mode and Effects Analysis. 15
IDE	Integrated Development Environment. 9
LSP	Language Server Protocol. ii, 1, 9
MDE	Model-Driven Engineering. 3
MDEG	Multi-Domain Engineering Graph. 3, 7–9, 13, 15
MDM	Multi-Domain Modeling. 1, 3, 13
MDM-CPPS	Multi-Domain Modeling for CPPS. ii, 3–9, 13–15
MvCM	Multi-view Change Management. 1, 2, 15
MvMF	Multi-view Modeling Framework. 2, 7
PPR	Product-Process-Resource. 2, 8–12
SUM	Single Underlying Model. 5
TMvMT	Traceable Multi-view Model Transformation. 1

Annex: List of references B

Bibliography

[1] VDI Guideline 3682 Formalised Process Descriptions., 2015.

[2] Marcelo Arenas, Claudio Gutierrez, and Juan F. Sequeda. Querying in the age of graph databases and knowledge graphs. page 2821–2828, 2021.

[3] Don S. Batory and Najd Altoyan. Aocl : A Pure-Java Constraint and Transformation Language for MDE. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 319–327, Setúbal, Portugal, 2020. SCITEPRESS.

[4] Florian Beetz and Simon Harrer. Gitops: The evolution of devops? *IEEE Software*, 39(4):70–75, 2022.

[5] Hugo Bruneliere, Jokin Garcia Perez, Manuel Wimmer, and Jordi Cabot. Emf views: A view mechanism for integrating heterogeneous models. In *International Conference on Conceptual Modeling*, pages 317–325. Springer, 2015.

[6] Irlán Grangel-González, Felix Lösch, and Anees ul Mehdi. Knowledge Graphs for Efficient Integration and Access of Manufacturing Data. In *ETFA*, volume 1, pages 93–100, 2020.

[7] Lavdim Halilaj, Irlán Grangel-González, Gökhan Coskun, Steffen Lohmann, and Sören Auer. Git4voc: Collaborative vocabulary development based on git. *Int. J. Semantic Comput.*, 10(2):167–192, 2016.

[8] Constantin Hildebrandt, Aljosha Köcher, Christof Küstner, Carlos-Manuel López-Enríquez, Andreas W. Müller, Birte Caesar, Claas Steffen Gundlach, and Alexander Fay. Ontology Building for Cyber-Physical Systems: Application in the Manufacturing Domain. *IEEE Trans Autom. Sci. Eng.*, 17(3):1266–1282, 2020.

[9] Eeva Järvenpää, Niko Siltala, Otto Hylli, and Minna Lanz. Implementation of capability matchmaking software facilitating faster production system design and reconfiguration planning. *Journal of Manufacturing Systems*, 53:261–270, 2019.

[10] István Koren, Felix Rinker, Kristof Meixner, Jasminka Matevska, and Jörg Walter. Challenges and opportunities of devops in cyber-physical production systems engineering. In *ICPS*, pages 1–6. IEEE, 2023.

[11] Stephan Krusche, Mjellma Berisha, and Bernd Bruegge. Teaching code review management using branch based workflows. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 384–393. Association for Computing Machinery, 2016.

[12] Kristof Meixner, Felix Rinker, Hannes Marcher, Jakob Decker, and Stefan Biffl. A Domain-Specific Language for Product-Process-Resource Modeling. In *ETFA*, 2021.

[13] László Monostori. Cyber-physical Production Systems: Roots, Expectations and R&D Challenges. *Procedia CIRP*, 17:9–13, 2014.

[14] Cosmina Cristina Ratiu, Wesley K. G. Assunção, Rainer Haas, and Alexander Egyed. Reactive links across multi-domain engineering models. In *25th MODELS*, pages 76–86. ACM, 2022.

[15] Felix Rinker, Sebastian Kropatschek, Thorsten Steuer, Kristof Meixner, Elmar Kiesling, Arndt Lüder, Dietmar Winkler, and Stefan Biffl. Efficient Multi-view Change Management in Agile Production Systems Engineering. In *24th ICEIS*, pages 134–141, 2022.

[16] Felix Rinker, Kristof Meixner, Laura Waltersdorfer, Dietmar Winkler, Arndt Lüder, and Stefan Biffl. Towards Efficient Generation of a Multi-Domain Engineering Graph with Common Concepts. In *ETFA*, pages 1–4, 2021.

[17] Felix Rinker, Laura Waltersdorfer, Kristof Meixner, Dietmar Winkler, Arndt Lüder, and Stefan Biffl. Traceable multi-view model integration: A transformation pipeline for agile production systems engineering. *SN Comput. Sci.*, 4(2):205, 2023.

[18] Miriam Schleipen, Arndt Lüder, Olaf Sauer, Holger Flatt, and Jürgen Jasperneite. Requirements and concept for plug-and-work. *at-Automatisierungstechnik*, 63(10):801–820, 2015.

[19] Jörn Guy Süß, Samantha Swift, and Eban Escott. Using devops toolchains in agile model-driven engineering. *Softw. Syst. Model.*, 21(4):1495–1510, aug 2022.

[20] Christian Tunjic and Colin Atkinson. Synchronization of projective views on a single-underlying-model. In *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 55–58, 2015.

[21] Birgit Vogel-Heuser, Markus Böhm, Felix Brodeck, Katharina Kugler, Sabine Maasen, Dorothea Pantförder, Minjie Zou, Johan Buchholz, Harald Bauer, Felix Brandl, et al. Interdisciplinary engineering of cyber-physical production systems: highlighting the benefits of a combined interdisciplinary modelling approach on the basis of an industrial case. *Design Science*, 6:e5, 2020.

[22] Andreas Wortmann, Olivier Barais, Benoit Combemale, and Manuel Wimmer. Modeling languages in Industry 4.0: an extended systematic mapping study. *Software and Systems Modeling*, 19(1):67–94, 2020.

[23] Yanan Xie and Yongsheng Ma. Well-controlled engineering change propagation via a dynamic inter-feature association map. *Research in engineering design*, 27(4):311–329, 2016.