JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Achieving Traceability and Reproducibility for Heterogeneous MBSE Processes using Ontology-Based Orchestration

Dirk Friedenberger[*], Jonas Henschel[†], Lukas Pirl[*], Can André Dautel[‡], and Andreas Polze[‡]
[*] *Hasso Plattner Institute*, *University of Potsdam*, Germany
[†] *Department of Computer Science*, *University of Technology Chemnitz*, Germany
[‡] *Institute of Aircraft Systems (ILS)*, *University of Stuttgart*, Germany

ABSTRACT The development of complex and safety-critical systems can benefit from robust and integrated MBSE processes. However, the ecosystem of MBSE methods, languages, and tools is fast-evolving and fragmented. MBSE processes are thus often practiced in isolation. This impedes integration, traceability, and reproducibility. This paper presents a process model for the formal description of transformation processes. We further specify an architecture for an execution environment that uses the process model as input. Together, the process model and the execution environment intend to enable automated, versioned, and traceable MBSE processes. The approach is being evaluated in the cross-domain *SQuIRRL* project, which concerns IT, railway, automotive, and aviation. The evaluation confirms dependency-consistent execution and reproducibility of heterogeneous MBSE transformation processes, thereby improving comparability and collaboration across domains. The metamodels, specification, and reference architecture are available as open source.

KEYWORDS Model-based systems engineering, Micro model, Transformation, Railway, Aviation, and Automotive

1. Introduction

Model-Based Systems Engineering (MBSE) is becoming increasingly important in the development of complex systems, particularly in safety-critical domains, such as aerospace, defense, automotive, and railway systems (Heydari 2023). According to (Technical Operations — International Council on Systems Engineering (INCOSE) 2007), "Model-based systems engineering is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases." In practice, MBSE promotes a common understanding of the system and supports interdisciplinary collaboration. MBSE enables model checking, supports code generation, and improves the traceability of requirements, design decisions, and test artifacts. As (Huldt & Stenius 2019) show, this leads to a better understanding of the properties and qualities of the system early in the design phase and throughout the entire life cycle. At the same time, MBSE facilitates compliance with industry standards, such as DIN EN 50716 (Deutsches Institut für Normung (DIN) 2023), as required in safety-critical domains. In these domains, modeling is identified as a measure to achieve a reliable, safe, and consistent development process. Thus, it is recommended or highly recommended across all Safety Integrity Levels (SILs) and supports requirements and architecture, software design and implementation, as well as analysis and overall testing.

In systems engineering, there is a broad ecosystem of formal methods with specialized, often mature languages, techniques, and tools. In practice, however, MBSE approaches are often used in isolation. In a "survey on formal methods and tools in railways" (Ferrari et al. 2019), the authors show that the B family (Atelier B, Event-B, ProB) and state machines

JOT reference format:
Dirk Friedenberger, Jonas Henschel, Lukas Pirl, Can André Dautel, and Andreas Polze. *Achieving Traceability and Reproducibility for Heterogeneous MBSE Processes using Ontology-Based Orchestration*. Journal of Object Technology. Vol. 25, No. 3, 2026. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2026.25.3.a23

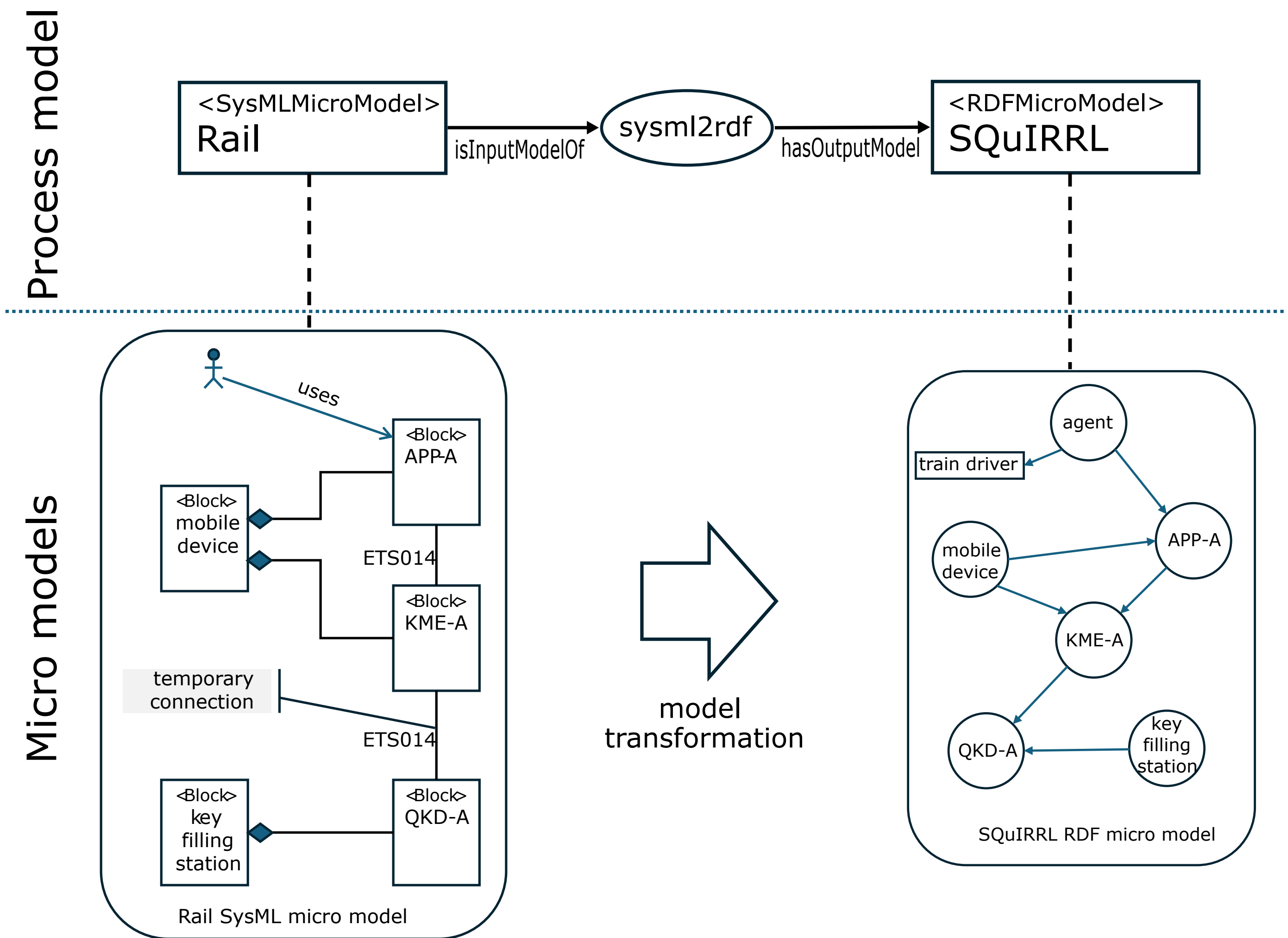

Figure 1 Exemplary illustration of the micro model transformation process. The process model describes how micro models (represented by rectangles) are transformed via transformation steps (represented by ellipses) into other micro models. Micro models describe a specific aspect or view of the system. The process model references the corresponding micro models (represented by rounded rectangles).

expressed in the Unified Modeling Language (UML) are particularly widespread in modeling and specification. Additionally, SCADE and Simulink/Stateflow dominate in development and code generation. NuSMV/nuXmv, SPIN, UPPAAL, CPN Tools, and UMC, among others, are used in formal verification and model checking. The integration of the different MBSE approaches is identified as a significant gap. Verification and Validation (V&V) toolchains are missing and tool interoperability is insufficient. This renders end-to-end development practically impossible. For example, there is no universally accepted formal framework that integrates MBSE modeling languages, such as SysML or UML, with formal verification tools, such as NuSMV or UPPAAL (Ferrari et al. 2019).

This supports the findings of (Sendall & Kozaczynski 2003) who showed in their paper "Model transformation: the heart and soul of model-driven software development", that the key challenge of model-driven development lies in the transformation of models. They further argue that these transformations must take place as an automated process according to clear transformation rules, enabling not only predefined but also user-defined transformations.

This raises the research question of how an automated, traceable, and reproducible modeling process can be implemented across domains.

To answer this question, we propose an approach that describes the process itself as a model, hereafter referred to as a *process model*. Building on the micro model and transformation concepts introduced in our work (Friedenberger et al. 2025), the process model describes the micro models involved (as views of the system, e.g., a component view) and the transformations between them (see Figure 1). The process model can thus also be seen as a graph, with micro models and transformations as nodes and semantic relations between them as edges. Additionally the process model is enriched with a configuration so that

implementations, e.g., transformations, can be executed in a containerized environment.

Contribution We develop a metamodel for the process model to describe the process of transformations and make the metamodel publicly available on *GitHub*.[1] In addition, we specify an architecture that enables the execution of the process model. For validation purposes, we provide a reference implementation based on this specification. Both the specification[2] and the reference implementation[3] are also published on *GitHub*.

The rest of this paper is structured as follows: Section 2 presents the nomenclature, defining key terms such as micro models, process models, and transformations. Section 3 introduces the process model and the architecture of the execution environment. We then present our use case in Section 4, followed by a prototypical reference implementation in Section 5. Section 6 evaluates the approach with respect to automation, reproducibility, traceability, and cross-domain applicability. Section 7 discusses related work and how existing solutions relate to our proposal. Section 8 discusses the results and limitations, and Section 9 concludes the paper.

2. Nomenclature

In this section, we define the central terms used throughout this work to ensure a consistent understanding of the concepts. Particular emphasis is placed on the distinction between micro models and the process model, as they form the foundation of the proposed approach.

2.1. Micro Model

A micro model is a small, loosely coupled part of a system model that represents a specific view or aspect of a system. The term is inspired by the concept of microservices in software engineering, where a monolithic application is decomposed into small, independent services. Examples of micro models can be found across the different domains of the *SQuIRRL* project, including SysML component diagrams, requirements documented in *LaTeX* templates, specifications in *Excel* sheets, and various artifacts, such as source code or documents. The concept of micro models and their associated transformations has been described in more detail in our prior work (Friedenberger et al. 2025).

2.2. Transformation

A transformation is a process that converts one or more input micro models into one or more output micro models. Transformations serve multiple purposes, such as converting between different micro models, extending an existing micro model, or producing artifacts required for development or verification. Importantly, transformations document the dependencies between input and output micro models, allowing the formalization of relationships and the derivation of execution order in the overall process.

[1] https://github.com/TheOpenMMTLab/mmut-rdf-model
[2] https://github.com/TheOpenMMTLab/mmut-execution-system-spec
[3] https://github.com/TheOpenMMTLab/mmut-orchestrator

2.3. Process Model

The process model formally describes the transformation process between micro models. It is composed of both micro models and transformations and is structured as a Directed Acyclic Graph (DAG). For practical execution, the process model can be extended to an executable process model by including *TaskDefinitions*. A *TaskDefinition* references containerized implementations for transformations and adapters for micro models.

2.4. Graphs

Based on the process model defined above, two graphs can be distinguished. The *Process Model Graph* contains micro models and transformations as nodes, with their dependencies as edges. Figure 5 illustrates an example of a Process Model Graph. From this graph, an *Execution Graph* is derived by performing a topological sort of the associated *TaskDefinitions*. A sequence of tasks results, that respects all dependencies and allows parallel execution. Figure 6 gives an exemplary Execution Graph.

2.5. Execution Environment

The execution environment is the system responsible for executing the Execution Graph, i.e., applying the transformations and model adapters. It ensures that the overall execution adheres to the dependencies defined in the process model, thereby enabling deterministic and automated process execution.

2.6. Containerization

Containerization refers to packaging transformations and adapters in isolated execution environments, such as *Docker* containers. By separating the model from implementation details, containerization enhances reusability, reproducibility, and portability, ensuring that transformations can be executed consistently across different environments.

2.7. Tools in MBSE Environments

Tools in MBSE environments are established, mature applications such as UPPAAL. These tools can be viewed as transformations between micro models, with their input and output represent the respective micro models. Integrating such tools within the process model allows them to participate in automated and traceable transformation chains.

3. Process Model and Proposed Architecture

In this chapter, we present the ontology of the process model and show how to model the transformation process. In Section 3.2, we specify an architecture that uses the process model as input.

3.1. Micro Model and Transformation Ontology

We outlined our concept of micro models and transformations in Section 2 and give details in a previous work (Friedenberger et al. 2025). The present work extends the concept of micro models and transformations with an RDF ontology. We published the sources of the ontology on *GitHub*[1] and visualized it in Figure 2. The ontology defines two abstract classes, *MicroModel* and *Transformation*, from which concrete specializations derive, such as *RDFMicroModel*, *SysMLMicroModel*,

and *PythonScriptTransformation*. Transformations are linked to their respective input and output micro models via the relations *isInputModelOf* and *hasOutputModel*. In addition, the relations *hasLooseCoupling* and *extendsModel* can be used to express dependencies between micro models that are not mediated by transformations.

In the current prototypical implementation, all transformations are realized as Python scripts. This is why *PythonScriptTransformation* is the only concrete subclass defined so far. The ontology is designed to be extensible, so that additional subclasses can be derived from the abstract *Transformation* class as needed. Examples include specializations for specific transformation technologies, such as XSLT, or tool integrations, such as for UPPAAL. We describe such specializations in Section 4.

As the definition in (Friedenberger et al. 2025) requires acyclicity, the resulting process model is a DAG (see Section 2). The constraints for the well-formedness of the process model are defined using the Shapes Constraint Language (SHACL). For example, each transformation must reference at least one input and one output micro model. The acyclicity of the graph is enforced through a SHACL constraint that detects loops via a SPARQL query on the transformation chain. The SHACL shapes[4] and their validation are documented in the ontology repository[1].

To extend the process model to an executable process model, a *TaskDefinition* can be added to transformations and micro models. For transformations, the task definition specifies how the transformation is executed. For micro models, it serves as an adapter that facilitates loading and storing of models. The *TaskDefinition* configures container properties such as the *Docker* image, the command to be executed, and any required environment variables (see Figure 3). We chose *Docker* because it provides the necessary abstraction between the model and the implementation details of the transformations, while also offering the flexibility needed to add further implementations without changing the execution environment.

For the process model, RDF/OWL has been chosen as the modeling formalism. As (Yang et al. 2019) show in their review of ontology-based systems engineering, formal semantics provide explicit, shareable, and reusable knowledge representation with well-defined domain concepts, terminologies, and relationships. RDF and OWL enable expressive queries via SPARQL, which can be used for traceability analysis, as demonstrated in Section 6. RDF and OWL also support consistency checking through reasoning and SHACL validation. As (Hillairet et al. 2008) discuss, EMF/Ecore provides analogous modeling capabilities and would have been equally suited for a model of this complexity. The choice of RDF/OWL has been primarily driven by the technology-agnosticism of RDF/OWL.

3.2. Architecture Specification

In this section, we specify an architecture for an execution environment, as there are several possible implementations. For example, pipeline-based build systems such as *GitLab* or any DAG-capable workflow engine, such as *Apache Airflow*,

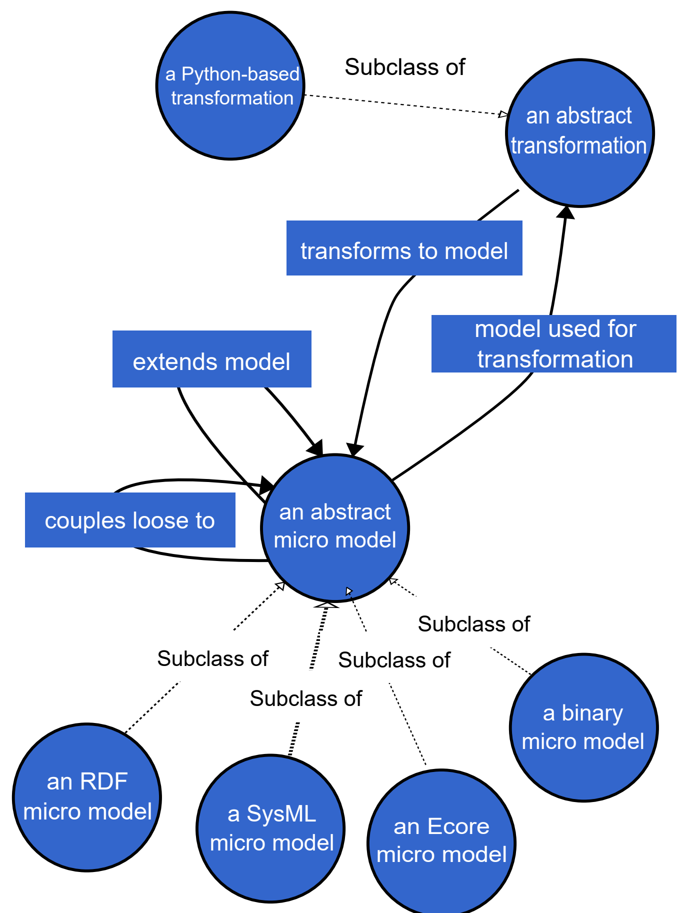

Figure 2 Excerpt from the ontology for micro models and transformations. Two abstract classes are shown, one for micro models and one for transformations. From these abstract classes, concrete classes derive. Models and transformations have semantic relations among each other.

[4] https://github.com/TheOpenMMTLab/mmut-rdf-model/blob/main/py_mmut_rdf/mmut-shapes.ttl

could orchestrate the executions. The specification aims to clearly define requirements and interfaces for the execution of transformations. We have also published the specification on *GitHub*[2].

The specification defines an ontology-driven architecture in which an execution environment executes containerized transformation and adapter tasks (see Figure 4). Task dependencies are defined correctly ordered in the DAG of the process model. The architecture follows three central principles. First, the ontology serves as a single source of truth by using micro models, transformations, and task definitions. The ontology also includes container properties and their environment, so that the container execution is unambiguous. Second, the execution order is derived from the process model DAG as described in Section 2. Third, the descriptive layer and the execution layer are strictly separated. The descriptive layer is defined by RDF and the ontology. The execution layer is implemented by container orchestration.

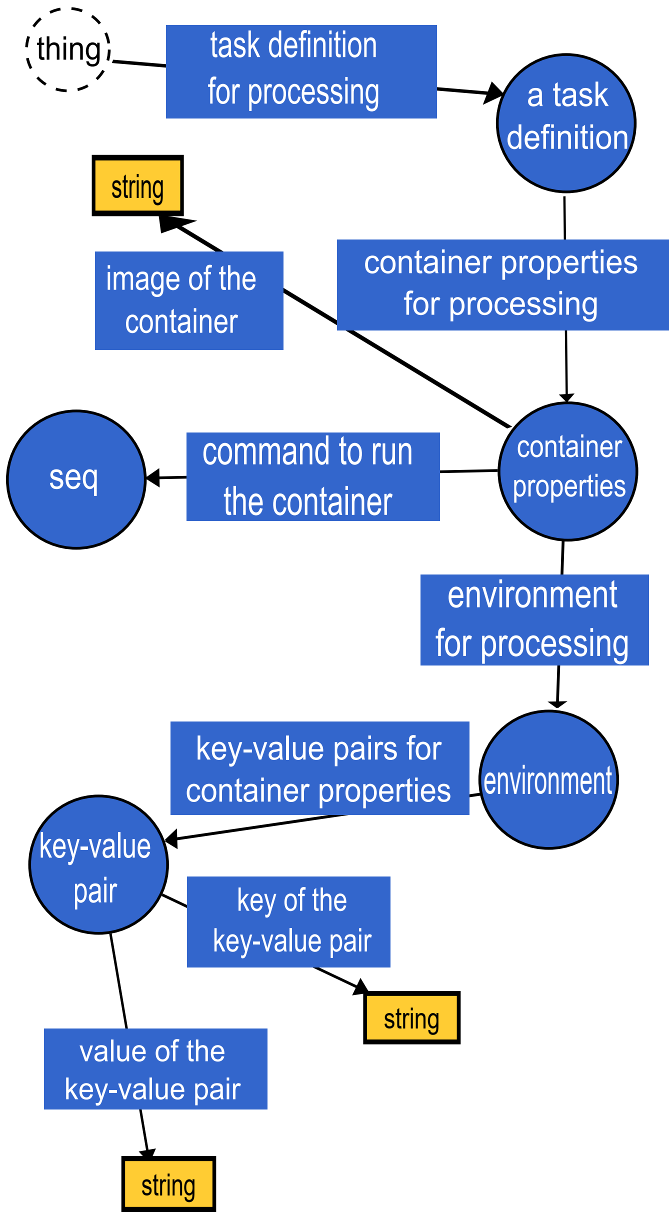

Figure 3 Extension of the micro model and transformation ontology. The class for a task definition specifies container properties, such as container image, command, and environment to reference a concrete implementation. This makes, e.g., transformations executable in a *Docker* environment.

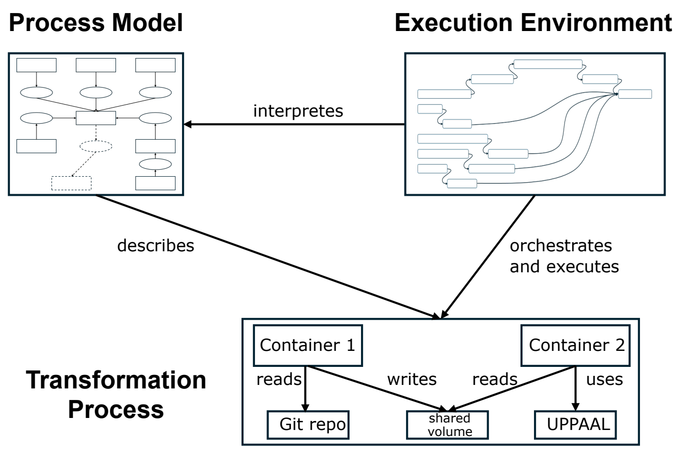

Figure 4 Architecture overview. The process model describes the transformation process. The execution environment interprets the process model and derives the Execution Graph, which determines the order in which containers are executed. In this example, Container 1 reads from the *Git* repository and writes the micro model to the shared volume. Container 2 then reads the micro model from the shared volume and uses UPPAAL for its verification.

4. SQuIRRL Use Case

The *SQuIRRL* project (Secure Quantum Infrastructure for Rail, Road, and Flight) is an interdisciplinary research initiative developing cost-efficient Quantum Key Distribution (QKD) solutions for the mobility domains railway, automotive, and aviation. QKD uses quantum mechanics to secure key exchange in the post-quantum era, protecting safety-critical communications against manipulation and eavesdropping. The project addresses domain-specific applications, including secure links between rail infrastructure and trains, vehicle-to-infrastructure communication in automotive, and authentication of avionics components in aviation. The consortium includes the Hasso Plattner Institute, Quantum Optics Jena, TU Berlin, TU Chemnitz, Hochschule In-

golstadt, HTW Dresden, Fraunhofer IIS, X-FAB, and University of Stuttgart.

Since different modeling tools are preferred in the respective domains, the researchers of *SQuIRRL* use a variety of techniques and tools to model the necessary requirements, architectures, and interaction scenarios. For example, in railway and automotive domain, use case and block definition diagrams are created using *Eclipse Papyrus*. Additionally, sequence diagrams are created in the railway domain using *PlantUML*. In comparison, the group in the avionics domain uses the *Open Avionics Architecture Model* (OAAM), which is based on *Ecore*, supplemented by the query language *Essential Object Query* (EOQ) and the self-developed visualization tool *eXtensible Graphical EMOF Editor* (XGEE) (Annighoefer et al. 2020). Across all domains, requirements and design decisions in the project context are documented using natural language and semi-formal *LaTeX* templates. In addition, some requirements are first recorded in *Excel* spreadsheets and then automatically transferred to *LaTeX*.

An objective is to establish a harmonized, tool-agnostic system representation (e.g., RDF) as a single source of truth. From this source, documentation and source code can be automatically generated where appropriate. Importantly, this homogeneous RDF/OWL representation does not imply a centralized model. While the micro models remain loosely coupled, RDF provides a unified semantic view. In addition, specialized formal models (e.g., UPPAAL for time-dependent behavior) are used to verify key system properties early and continuously in the model. The results can also be fed back into the harmonized model and into the V&V tool chain.

Specifically, we can convert the heterogeneous models into a common homogeneous representation using transformations (e.g., SysML to RDF). Further transformations are required to derive specialized model formats from the common representation. For example, for the verification of system properties with UPPAAL, a transformation can be applied to generate a UPPAAL timed automata model from the state machines defined in the system models. The invocation of UPPAAL itself can in turn be regarded as a transformation that takes the UPPAAL model as input. The resulting output can then be used in another transformation to generate documents needed for the certification process. Each of these models is a micro model in the process model. We have added the corresponding transformations between the models, as depicted in Figure 5.

This approach enables interoperability and integration across model boundaries, including linking and traceability. Choosing a representation, such as the Resource Description Framework (RDF) and the Web Ontology Language (OWL), establishes the basis for expressive queries using the RDF Query Language (SPARQL). RDF/OWL also enables inference, reasoning, and supports consistency checks with the SHACL.

5. Implementation

To verify feasibility, we have developed a prototype reference implementation. We use *Prefect*[5], a lightweight, *Python*-based orchestration framework, to orchestrate the execution of adapter

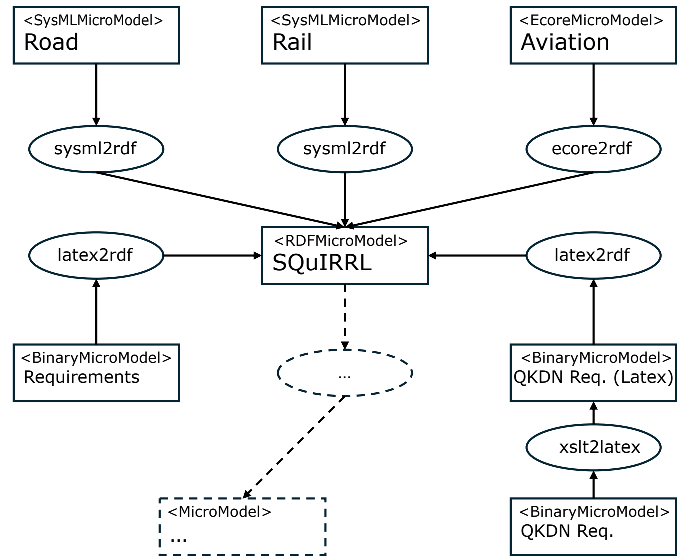

Figure 5 Excerpt from the modeling of the transformation process in the *SQuIRRL* project. Rectangles represent micro models, while ellipses represent the transformations between them. The dashed lines indicate future extensions, such as the transformation into a model that can be used for UPPAAL.

and transformation implementations. To determine the execution order, we use *NetworkX*[6], a *Python* package that can calculate topological sorting in a DAG. We have published our implementation on *GitHub*[3].

Prefect is based on the two central concepts *tasks* and *flows* for defining data pipelines and automations. A *task* represents a single processing step within the orchestration. In our implementation, this concept is used to start and monitor *Docker* containers that are responsible for transformations and model adapters. The use of *Docker* creates a clear level of abstraction, so that the orchestration itself is unaware of the business logic within the container. In this way, a single *task* is sufficient to execute all implementations. The configuration of the container is taken from the *TaskDefinition* of the process model. This configuration might include, e.g., the name of the image to use, the command to run, and environment variables.

The *flow* concept is used in *Prefect* to define entire workflows. In our implementation, it is used to automatically generate a complete workflow from a given process model. To determine the execution order of the individual *tasks*, a topological sort is first calculated using the *topological_sort* function of the *NetworkX* package. This sequence makes it possible to correctly define the dependencies between the implementations in the *flow*. *Prefect* then executes the *tasks* in the correct order, allowing parallel execution as long as all dependencies are met. To provide the implementations with the necessary models, a shared file storage is used, which is mapped into the corresponding containers.

In the *SQuIRRL* project, we use this implementation of orchestration to execute the transformations automatically and in the correct order. We developed the individual transformations,

[5] https://www.prefect.io/

[6] https://networkx.org/

such as SysML to RDF, as *Python* scripts and then containerized them so that they could be used in the execution environment. As mentioned, implementations can also be linked to the micro models. We use these as adapters to load models (e.g., from *Git*) or save models (e.g., to *Git* or in an RDF triple store). In Figure 6, the process *flow* in the *Prefect* UI shows the successful execution of adapters and transformations in the order determined by their dependencies.

The presented implementation provides the foundation for the following evaluation, in which we examine to what extent the architecture fulfills the targeted properties – automation, reproducibility, traceability, and domain independence.

6. Evaluation

This section describes the concept evaluation of our RDF ontology for micro models and transformations. We describe evaluation questions, the evaluation methodology, and the process model used for the evaluation. In the course of giving the evaluation results, we refer back to our evaluations questions.

6.1. Questions and Methodology

The evaluation is conducted as a technical case study based on the *SQuIRRL* project. We investigate which steps of the transformation process can be automated, whether the results are reproducible given identical inputs, whether the transformation chain from input to output can be traced through the Process Model Graph, and whether the approach can be applied across domains with heterogeneous tools.

The evaluation is based on four criteria:

- **Automation:** We determine which steps of the transformation process are executed automatically by the orchestration and which still require manual intervention.
- **Reproducibility:** We examine whether the overall process produces identical outputs given identical inputs. Additionally, we examine whether individual steps can be reproduced in isolation, in particular in case of failures.
- **Traceability:** Ramesh and Jarke (Ramesh & Jarke 2001) distinguish four types of traceability links: Satisfaction, Dependency, Evolution, and Rationale. We examine which of these types are covered by the process model and demonstrate traceability through a concrete error scenario in the *SQuIRRL* project.
- **Cross-domain applicability:** We examine whether the execution environment is domain-neutral, that is, whether it has no explicit dependencies on a specific discipline, and whether additional domains can be integrated without modifications to the orchestration.

6.2. Process Model of the Case Study

The evaluation is based on the process model of the *SQuIRRL* project, which describes the transformation chains of the involved domains railway, automotive, and aviation. The process model comprises heterogeneous input models in different formats, including SysML models via *Eclipse Papyrus*, *LaTeX* documents, and *Excel* spreadsheets. All those input models are transformed into a common RDF representation.

The current process model contains seven micro models, of which three are of type *SysMLMicroModel*, three of type *BinaryMicroModel*, and one of type *RDFMicroModel*. It comprises six transformations, all realized as *PythonScriptTransformation*, and 13 *TaskDefinitions*, of which six are assigned to transformations and seven to adapters for micro models. In total, seven different container images and four model formats are used: SysML, *LaTeX*, *Excel*, and RDF/OWL. The railway domain is currently covered with implemented transformations, while automotive and aviation are currently included as placeholders in the process model. Listing 1 shows an excerpt of the process model in *Turtle* syntax.

In addition to the SQuIRRL process model, a synthetic test model has been created to evaluate the orchestration with a significantly larger graph. This model contains 100 transformations, 108 micro models, and 208 *TaskDefinitions*, distributed across two connected components with a maximum path length of 15 transformation steps. The transformations use a simulation component to which configurable functions and execution times can be assigned. Table 1 compares the graph properties of both models.

Table 1 Graph properties and total execution time of the SQuIRRL process model and the synthetic test model.

Property	SQuIRRL	Synthetic	Fact.
Transformations	6	100	16.7
Micro Models	7	108	15.4
Task Definitions	13	208	16
Connected Components	1	2	2
Longest Path	2	15	7.5
Total Execution Time	17.82 s	595.96 s	33.4

6.3. Results

Automation In the SQuIRRL process model, the following steps are executed automatically:

- Computation of the execution order from the dependencies modeled in the process model.
- Loading of input models via adapters, for example from *Git* repositories or *GitLab*.
- Execution of all six transformations, including SysML to RDF, *LaTeX* to RDF, and XLSX to *LaTeX*.
- Storing of results via adapters, for example by pushing to an RDF triple store or to a *Git* repository.

The following steps still require manual intervention:

- Creation and maintenance of models in GUI-based tools such as *Eclipse Papyrus*.
- Provisioning of models in a version control system so that adapters can access them.
- Creation and maintenance of the process model itself.
- Containerization of new transformations.

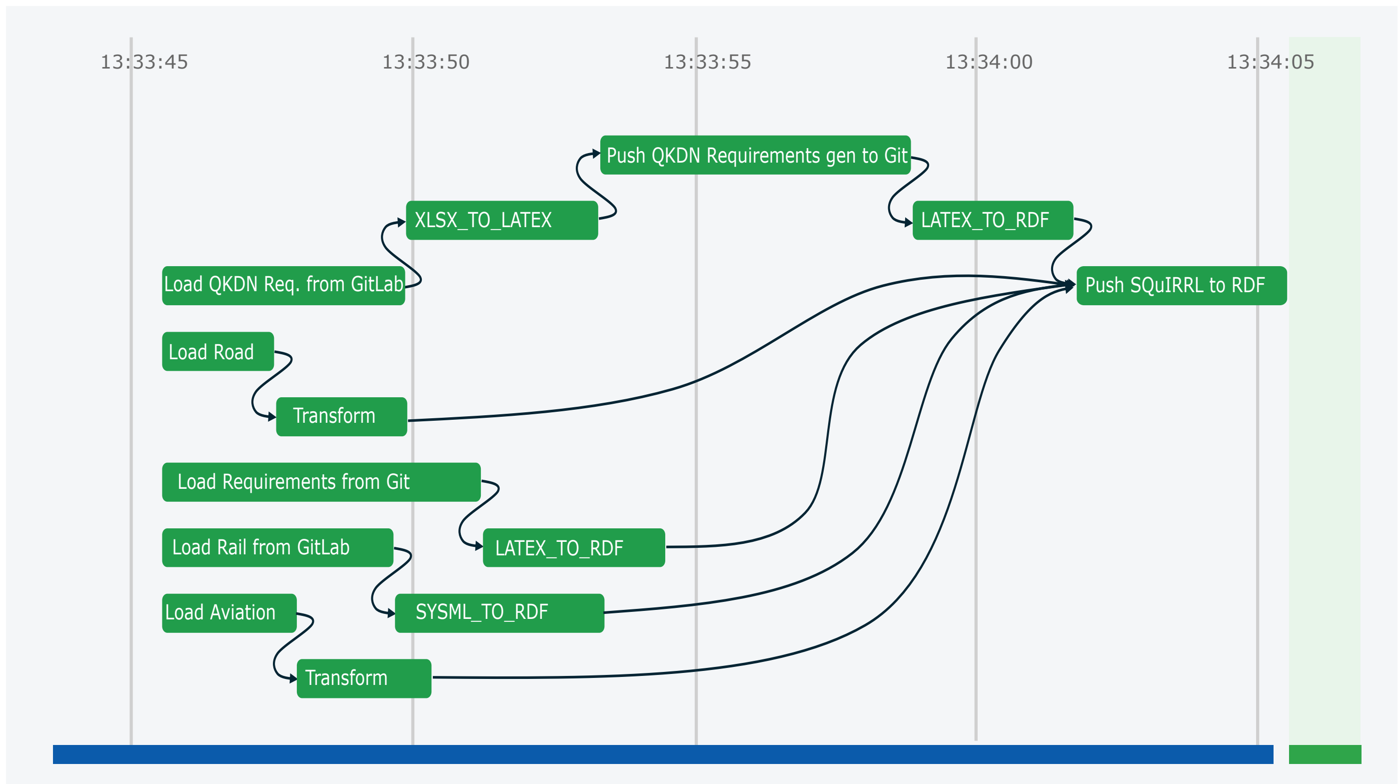

Figure 6 Dependency-driven *Prefect* process flow showing the successful (green) executions of adapters and transformations.

The automated steps cover the recurring execution process, from loading the input models, computing the execution order, and executing all transformations, to storing the results. The manual steps concern the ongoing development of models in the respective tools, the maintenance of the process model when the transformation process changes, and the containerization of new transformations. The approach therefore fully automates the execution of the process, but not the modeling itself.

Reproducibility Reproducibility of results is achieved through explicit versioning of all involved artifacts. Micro models are loaded from a version control system and assigned to a specific commit. The implementations of transformations and adapters are available as container images and can be referenced via immutable digests. Listing 1 illustrates this: the *Git* adapter references both a specific commit hash via the `-ref` parameter and a container image via its SHA digest. Alternatively, Semantic Versioning[7] tags for container images or *Git* tags can be used for better readability, while the principle of explicit versioning remains the same. Under identical conditions, in particular with fixed model revisions and image digests, the execution process is deterministic.

Furthermore, individual steps of the process can be reproduced in isolation. The shared volume is implemented such that the files of each run are stored in a separate directory and preserved. This makes the input data of a failed step available for isolated reproduction, as shown in Listing 2. This facilitates error analysis and debugging of individual transformations.

[7] https://semver.org/

For empirical verification of reproducibility, the pipeline has been executed twice with identical inputs, each time on a different machine and operating system. The first run has been executed under Windows 11 and the second run under Debian GNU/Linux 6.1. The SHA256 checksums of all generated artifacts from both runs are identical, as Table 2 shows. The matching checksums across operating systems confirm that containerization ensures complete isolation of the execution. A script for computing and documenting artifact checksums is available in the reference implementation[3]. This script can also be used during operation to document the checksums of generated artifacts and thereby demonstrate reproducibility over time.

Table 2 SHA256 checksums of selected artifacts from two independent pipeline runs on different operating systems. Checksums are truncated for readability.

Artifact	Run 1	Run 2	Match
	flow-petite-honeybee	*flow-cunning-reindeer*	
	Windows 11	Debian Linux 6.1	
rail.uml	`13feec58...`	`13feec58...`	✓
rail.ttl	`8d928f27...`	`8d928f27...`	✓
requirements.ttl	`bd96db59...`	`bd96db59...`	✓
Aggregate	`8473c7a4...`	`8473c7a4...`	✓

```
@prefix MMUT:    <http://frittenburger.de/ontology/mmut#> .
@prefix SQuIRRL: <https://frittenburger.de/2022/11/EULYNX#>
    .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-
    ns#> .

# --- Micro Model (Input) ---
SQuIRRL:BinaryMicroModel-Requirements
    a MMUT:BinaryMicroModel ;
    rdfs:label "Requirements" ;
    rdfs:comment "Requirements Model" ;
    MMUT:hasTaskDefinition
        SQuIRRL:TaskDefinition-Load-Requirements-from-Git ;
    MMUT:isInputModelOf
        SQuIRRL:PythonScriptTransformation-d .

# --- Micro Model (Output) ---
SQuIRRL:RDFMicroModel-SQuIRRL a MMUT:RDFMicroModel ;
    rdfs:label "SQuIRRL" ;
    rdfs:comment "SQuIRRL System Model" ;
    MMUT:hasTaskDefinition
        SQuIRRL:TaskDefinition-Push-SQuIRRL-to-RDF-Store .

# --- Transformation ---
SQuIRRL:PythonScriptTransformation-d
    a MMUT:PythonScriptTransformation ;
    rdfs:comment "Create Requirements Model" ;
    MMUT:hasOutputModel SQuIRRL:RDFMicroModel-SQuIRRL ;
    MMUT:hasTaskDefinition
        SQuIRRL:TaskDefinition-LATEX_TO_RDF-d .

# --- Task Definitions ---
SQuIRRL:TaskDefinition-Load-Requirements-from-Git
    a MMUT:TaskDefinition ;
    rdfs:label "Load Requirements from Git" ;
    MMUT:hasContainerProperties
        SQuIRRL:ContainerProperties-Load-Requirements .

SQuIRRL:TaskDefinition-LATEX_TO_RDF-d
    a MMUT:TaskDefinition ;
    rdfs:label "LATEX_TO_RDF d" ;
    MMUT:hasContainerProperties
        SQuIRRL:ContainerProperties-LATEX_TO_RDF-d .

# --- Container Properties (Adapter) ---
SQuIRRL:ContainerProperties-Load-Requirements
    a MMUT:ContainerProperties ;
    MMUT:image "frittenburger/git-adapter@sha256:eb84..." ;
    MMUT:hasCommandSequence [ a rdf:Seq ;
        rdf:_1 "./get.sh" ;
        rdf:_2 "--repo" ;
        rdf:_3 "https://git@git.overleaf.com/..." ;
        rdf:_4 "--input-path" ;
        rdf:_5 "_inhalte.tex" ;
        rdf:_6 "--output-path" ;
        rdf:_7 "{{resolve:system:modelpath}}requirements.
    tex" ;
        rdf:_8 "--ref" ;
        rdf:_9 "d3bed6..." ] ;
    MMUT:hasEnvironment
        SQuIRRL:Environment-Load-Requirements .
```

Listing 1 Excerpt from the SQuIRRL process model in *Turtle* syntax, showing one input micro model, one output micro model, one transformation, and the container properties of the *Git* adapter with commit hash and image digest. Hashes and links are truncated for readability.

```
docker run -v ./shared/petite-honeybee/models:/models \
  frittenburger/git-adapter@sha256:eb84... \
  ./get.sh --repo https://... \
  --ref d3bed6...
```

Listing 2 Isolated reproduction of a single transformation step using Docker. Hashes and links are truncated for readability.

Traceability With respect to the reference model of Ramesh and Jarke (Ramesh & Jarke 2001), the Process Model Graph covers two of the four traceability link types. The process model explicitly captures which micro models serve as inputs to which transformations and which micro models result from them. These directed dependencies correspond to *Dependency Links*. Because micro models are linked through transformations, the evolution of models can be traced along the transformation chain, which corresponds to *Evolution Links*. Because the process model is structured as a DAG, the complete path forward and backward can be determined for each node in a finite number of steps through graph traversal.

It should be noted that this traceability applies at the process level, that is, at the level of entire micro models and transformation steps. The approach of (Ramesh & Jarke 2001) additionally proposes *Satisfaction Links* and *Rationale Links*. *Satisfaction Links* describe whether a model fulfills a specific requirement. *Rationale Links* document why a particular transformation has been chosen. Both are not covered by our work so far. Likewise, fine-grained traceability at the object level within models is not yet addressed.

To demonstrate traceability in an error scenario, a SPARQL query has been executed on the SQuIRRL process model. The scenario assumes that an error is discovered in the generated SQuIRRL RDF model, for example during a SHACL validation. The traceability question is: Which input models and transformations contributed to this faulty model? Listing 3 shows the SPARQL query, which performs a backward traversal in the Process Model Graph.

```
SELECT ?inputLabel ?transComment WHERE {
  ?input MMUT:isInputModelOf ?trans .
  ?input rdfs:label ?inputLabel .
  ?trans  MMUT:hasOutputModel
          sq:RDFMicroModel-SQuIRRL .
  ?trans  rdfs:comment ?transComment .
}
ORDER BY ?inputLabel
```

Listing 3 SPARQL query for backward traceability: Given an output model, find all directly contributing input models and transformations. The query traverses one level; by iteratively applying it to the returned input models, the complete provenance chain can be reconstructed. Prefix declarations omitted for brevity.

The result of the iterative application is shown in Table 3. In the first iteration, five input models are identified, each contributing to the *SQuIRRL* RDF model via one transformation. In the second iteration, the model *QKDN Requirements* is additionally identified as a predecessor of *QKDN Requirements gen*. This allows reconstructing the complete provenance chain of the faulty model and narrowing down the root cause. Termination is guaranteed by the acyclicity of the graph.

Cross-Domain Applicability The abstraction through transformations and adapters (see Section 3) forms the basis for the domain neutrality of the execution environment. The orchestration only knows *TaskDefinitions* with container images and

Table 3 Result of the iterative backward traceability query starting from the SQuIRRL RDF output model. Depth indicates the iteration level.

Depth	Input Model	Transformation	Output Model
1	Aviation	Create Aviation System Model	SQuIRRL
1	QKDN Req. gen	Create Requirements Model	SQuIRRL
1	Rail	Create Rail System Model	SQuIRRL
1	Requirements	Create Requirements Model	SQuIRRL
1	Road	Create Road System Model	SQuIRRL
2	QKDN Req.	Create QKDN Req. Model	QKDN Req. gen

commands, while the domain-specific logic is fully encapsulated in the containers. Integrating new domains only requires implementing domain-specific transformations and adapters as containers. Modifications to the orchestration are not needed.

As described by the process model, the railway domain is currently covered with implemented transformations, while automotive and aviation are included as placeholders in the process model. For the aviation domain, whose ecosystem is described in Section 4, there are no fundamental obstacles to integration. The models used there, which are based on OAAM and *Ecore*, are file-based and can be loaded via adapters analogous to the already implemented *Git* adapters. The connection to object stores can be considered analogous to the already implemented connection to the RDF triple store. For the automotive domain, the integration has not yet started, but the same adapter-based approach applies.

Cross-domain applicability has so far been argued on a conceptual basis. The full integration of the automotive and aviation domains is planned as part of the ongoing *SQuIRRL* project.

6.4. Stakeholder Feedback

The presented approach has been discussed in research colloquia at the University of Stuttgart and at the Chemnitz University of Technology with researchers from the involved domains. Initial feedback on practical applicability and domain-specific requirements has been discussed. A systematic collection and analysis of this feedback as well as its incorporation into the approach are planned as future work.

7. Related Work

In the following, we compare our approach with existing solutions that address the integration of heterogeneous models in MBSE environments. We then summarize the comparison in Table 4. Finally, we discuss identified gaps in current MBSE processes.

The paper (Walter et al. 2019) proposes an architecture that uses graph-based design languages to transform classic, highly manual MBSE workflows into a digitalized, machine-executable process. Despite the use of UML and SysML, models are usually linked manually due to a lack of interfaces between domain-specific tools. Walter et al. therefore propose a generic, abstract centralized model that can be translated into any domain-specific model. This model is used to create an abstract digital twin of the developed product and to derive a finite state machine for the system. This state machine is consistent with the specified system requirements and can be tested as an executable model. Automation is achieved here through automatic transformations, as in our approach, and through the automatic execution of the system model. However, Walter et al.'s approach uses a centralized and generic model to achieve a link between the domains. Our approach differs in this respect, as we do not require such a centralized model, but neither do we exclude it. Thus, our approach offers a more flexible solution. We implement the link between domains via the proposed process model.

In (Zhang et al. 2025), the authors address the MBSE challenges of high entry barriers, increasing task complexities, strong ecosystem fragmentation, and fast technology evolution. Prospects for the adoption and usability of MBSE are seen in superordinate tooling, especially in automation through Artificial Intelligence (AI). The authors propose the concept of an MBSE Co-Pilot that can help in model comprehension, model development, and model management. Cross-cutting with these capabilities, the authors provide a research road map concerning data readiness, traceability, and trustworthiness. These aspects are assessed in a small case study with a Co-Pilot plugin for a commercial modeling tool, attesting data readiness is almost given today. Abstracting from their concrete concept, the authors propose autonomy levels for MBSE support tools in analogy to autonomy levels in other domains, e.g., robotics. These levels reach from rule-based automation tasks (level 1), over modeling assistance with feedback (level 2), to decision suggestion with contextual understanding (level 3). Our approach to a process model based on the *Micro models and Transformations* approach in (Friedenberger et al. 2025) is aligned with the motivation presented in (Zhang et al. 2025). We especially share the assumption that superordinate tooling and automation is beneficial. However, we deliberately abstain from using AI in order to be able to achieve reproducibility, explainability, verifiability, and approvability. Instead of an approach based on AI, we favor the proven approaches of modularization (micro models), abstractions (ontologies), automation (transformations), and continuity (workflow engine). Nevertheless, these differences are not inherently incompatible. More formalized relations between models and more formalized processes could

be of help for an MBSE Co-Pilot as proposed by Zhang et al. We expect this helpfulness especially for the Co-Pilot's understanding of context, of domain-specifics, of dependencies, but also for tracing behavior and data.

The work by (Meng et al. 2025) presents the RACK Integrated Certification Environment (RITE), an Eclipse-based tool that combines ontology development, curated evidence collection, and automatic synthesis of assurance cases and compliance reports in a unified environment. The goal is to make the certification and assurance of safety-critical systems more efficient, especially in the context of aviation standards such as DO-178C and ARP4754A. RITE uses semantic models and ontologies to harmonize heterogeneous data. We also use this approach in the *SQuIRRL* project to harmonize the various domain models, but our approach is not limited to this. While RITE relies on a central knowledge base (the Rapid Assurance Curation Kit, RACK) and a uniform ontology, we link the domains involved using a process model-based approach with loosely coupled micro models and transformations. In this way, we avoid a centralized, generic model structure. RITE focuses on the automated synthesis of assurance cases and compliance reports from the collected development artifacts. However, it does not cover the entire life cycle. In particular, it lacks integrations with requirements engineering and risk analysis. Our approach, on the other hand, is more generic and allows the integration of any models and transformations. This makes it more flexible and presumably easier to transfer to different use cases.

The Vitruvius framework (Klare et al. 2021) addresses the consistency of heterogeneous models in view-based system development. It introduces the concept of a Virtual Single Underlying Model (V-SUM), which combines existing models and preserves their consistency through delta-based Consistency Preservation Rules (CPRs). When a developer modifies one model, the CPRs automatically propagate the changes to the related models. Both Vitruvius and our approach share the same starting point, namely the fragmentation of tools, languages, and models in MBSE environments. However, they pursue different goals. Vitruvius focuses on maintaining consistency between models during interactive editing at the level of individual model elements. Our approach focuses on the orchestration and reproducibility of transformation processes across heterogeneous models at the level of entire micro models and transformation steps. Furthermore, Vitruvius supports bidirectional transformations through its Reactions and Mappings languages, whereas our process model is structured as a DAG with strictly unidirectional transformations. This avoids the challenge of bidirectional consistency preservation but does not provide real-time synchronization between models. The two approaches can therefore be seen as complementary rather than competing.

In the paper (Jackson et al. 2016), the authors argue that MBSE often shows only a snapshot of the Software Engineering (SE) process. The maturity level of requirements and designs, changes, decision-making reasons, and the status of the SE process are hardly visible or comprehensible in the model. This hinders progress, quality assurance, and collaboration in large projects. As a solution, they propose expanding the model vocabulary and tool infrastructure to include explicit maturity tracking for requirements and design artifacts. Our approach does not offer direct maturity tracking. However, explicit modeling of the process and the use of versioned models and versioned containers for implementations (e.g., transformations) result in a reproducible version status of the process. The development status can thus be tracked using the versioning described above.

Common pain points of MBSE projects and how they are similar to the problems software engineering had to master some decades ago are described in (Papke et al. 2023). From their explanations they conclude that MBSE projects will benefit from considering life cycle management of models more as a typical software engineering life cycle then as the widely established waterfall approach to document management. Papke et al. put a particular emphasis on quality management and modularization of models as well as iterative/agile modeling processes. They also mention poor tool performance as a common pitfall for many MBSE projects. In the paper (Papke et al. 2023), the authors provide an "actionable set of best practices" for model development in MBSE projects. While our approach cannot directly influence model quality, e.g., through validity checks, it is possible to automatically reevaluate generated models after reiterating over some input models. This automatic reevaluation through the orchestration feature can also be used to automate model testing and verification mentioned in (Papke et al. 2023). Moreover, due to the modular nature of *Micro models and Transformations* approach, models can be kept small, thus dealing with poor performance of modeling tools. To aid with model modularization our transformation concept can act as a transpiler from one output model to another input model if tools cannot directly interface with each other. In addition, the high degree of adaptability of our approach benefits the development of agile models. Therefore, we argue that our approach of model the process based on *Micro models and Transformations* can help with the application and automation of most of Papke et al.'s best practices.

Table 4 summarizes the comparison along eight criteria: integration of *heterogeneous tools*, *traceability* of transformation chains, *reproducibility* of results, *automation* of the transformation process, the *modeling language* and *framework* used, the *model architecture*, and *cross-domain* applicability. It shows that the compared approaches differ in their focus. The Vitruvius framework (Klare et al. 2021) provides strong consistency preservation through bidirectional CPRs, while (Walter et al. 2019) and RITE (Meng et al. 2025) rely on a central model. (Zhang et al. 2025) pursues a complementary direction through AI-based automation. Our approach combines ontology-based process description with containerized execution, focusing on traceability and reproducibility of transformation processes.

8. Discussion

The proposed approach addresses one of the challenges in MBSE environments, as highlighted by (Ferrari et al. 2019; Zhang et al. 2025). It is characterized by the fragmentation of tools, languages, and domain-specific modeling practices. Instead of using a centralized meta-model or a unified modeling

Table 4 Comparison of related approaches. ✓ = supported, ○ = partially, – = not described.

Criterion	Our approach	Walter et al.	Zhang et al.	RITE	Vitruvius
Heterogeneous tools	Containerization	Domain-specific models	–	–	EMF metamodel
Traceability	✓	–[a]	○[b]	✓	✓
Reproducibility	✓	–	–	–	○[c]
Automation	✓	✓	○[d]	✓	✓
Modeling language	RDF/OWL	UML/SysML	–	OWL	EMF/Ecore
Framework	not prescribed	–	Commercial[e]	Eclipse	Eclipse/EMF
Model architecture	Micro models	Central model	–	Central KB	V-SUM (modular)
Cross-domain	○[h]	○[f]	○[b]	Aviation[g]	✓

[a] Implicit through transformation chain, not explicitly described. [b] Mentioned in research roadmap, not implemented. [c] Delta-based, but execution order of CPRs may affect result in worst case. [d] AI-based automation envisioned, autonomy levels 1–3 defined. [e] Concrete tool not named in paper. [f] Possible via central model, not explicitly demonstrated. [g] Focus on DO-178C and ARP4754A. [h] Railway covered with implemented transformations, automotive and aviation as placeholders.

core, our approach enables integration through an explicit process model that loosely couples heterogeneous micro models and their transformations.

As noted in Section 1, DIN EN 50716 (Deutsches Institut für Normung (DIN) 2023) applies to the safety-critical domains addressed in this work. This has implications for the execution environment itself. The approach has been developed with the requirement that the execution environment must be suitable for safety-critical systems. According to the standard, the execution environment is expected to be classified as a tool of class T2 or T3. For this reason, a specification has been developed first, and the prototype serves to demonstrate the targeted properties. The properties traceability and reproducibility, which are particularly relevant for certification, are therefore central to the approach.

Reproducibility is achieved through containerization and explicit versioning of models and container images, as demonstrated empirically in the evaluation. The current evidence is based on two runs on different operating systems. Broader validation across environments and teams remains future work.

The relationships between micro models and transformations are represented semantically at the ontology level. Each transformation and each micro model has defined inputs, outputs, and semantic relations. SPARQL queries on the process model can be used to find transformations and analyze their dependencies, as demonstrated in Section 6. However, containerization means that the orchestration does not know the implementation details of individual steps. Transformations are executed as black boxes. This abstraction is intentional, as it provides a uniform interface for all transformations and avoids dependencies between the orchestration and the technologies used. New transformations can be added to the process without modifying the execution environment. To improve the comprehensibility of individual transformations, metadata annotations such as `rdfs:label` and `rdfs:comment` can be used in the ontology to describe their purpose and behavior in a human-readable way.

A consequence of this design decision is that traceability applies at the process level, that is, at the level of entire micro models and transformation steps. Fine-grained traceability at the object level within models is not available. Likewise, there is currently no temporal traceability of changes over longer development cycles.

In addition to the transformation-based relations *isInputModelOf* and *hasOutputModel*, the ontology defines the relations *extendsModel* and *hasLooseCoupling* between micro models. These capture dependencies that go beyond the data flow of transformation chains, such as semantic cross-dependencies between models of different domains. In the current SQuIRRL process model, these relations are not yet used. Their systematic application could extend traceability with an additional dimension and improve impact analysis for cross-domain changes. Similarly, the *Satisfaction Links* and *Rationale Links* identified as not covered in the evaluation could be addressed through additional semantic relations in the ontology, which would also improve the comprehensibility of transformations executed as black boxes.

Regarding scalability, the topological sort used to derive the execution order has a time complexity of $O(V + E)$ (Kahn 1962). This is efficient even for large graphs. The execution of a synthetic test model of approximately factor 16 in size completed in approximately factor 33 in time (see Table 1), indicating that the orchestration scales to process models significantly larger than the current SQuIRRL model. Potential bottlenecks for further scaling include the *Docker* container startup overhead for many small transformations and the shared volume as an I/O bottleneck for large models. A possible mitigation would be incremental execution that only re-runs affected subgraphs after changes to the process model, which is not yet implemented and remains future work.

Further limitations concern automation and debugging. Some tools require manual triggers or the use of graphical interfaces, which limits full automation. In addition, testing and debugging capabilities within the orchestrated system are reduced due to the high level of abstraction, although local reproduction of individual steps remains possible.

The approach facilitates collaboration across domain boundaries, as demonstrated in the *SQuIRRL* project involving railway, automotive, and aviation sectors. Nevertheless, the challenge lies in standardizing and consistently managing artifacts across these domains. Each of the domain models requires an individual transformation that maps the domain model to a common homogeneous representation. This could be simplified by specifying the methods and tools used within a project. However, these restrictions do not contradict our approach, but rather form organizational framework conditions that can help in the implementation of cross-domain scenarios.

9. Conclusion

In this work, we propose an ontology-based process model that formally describes transformation processes between heterogeneous micro models. We specify an architecture for a containerized execution environment and provide a reference implementation. The process model, the specification, and the reference implementation are published as open source on *GitHub*.

The evaluation demonstrates that the approach enables automated, reproducible, and traceable execution of heterogeneous transformation processes. Cross-domain applicability is supported by the design of the process model, which describes micro models and transformations at an abstract level, while the domain-specific logic is encapsulated in containers. The practical applicability is demonstrated in the *SQuIRRL* project.

Overall, the architecture shows how heterogeneous model landscapes can be integrated into traceable and reproducible development processes through a process-oriented metamodel combined with containerized execution.

Future work includes the derivation of a QKD prototype from the homogeneous model representation in the *SQuIRRL* project, the standardization of the container interface for model injection, and the extension of traceability through additional semantic relations in the ontology, as discussed in Section 8.

Acknowledgments

The presented results are part of the work in the research project *SQuIRRL* (contract code: 16KIS2165), which was supported by the German Federal Ministry for Research, Technology and Space (BMFTR).

References

Annighoefer, B., Brunner, M., Schoepf, J., Luettig, B., Merckling, M., & Mueller, P. (2020). Holistic IMA platform configuration using web-technologies and a domain-specific model query language. In *2020 aiaa/ieee 39th digital avionics systems conference (dasc)* (pp. 1–10). doi: 10.1109/DASC50938.2020.9256726

Deutsches Institut für Normung (DIN). (2023). *Bahnanwendungen – anforderungen für die softwareentwicklung* (No. DIN EN 50716:2023). Norm. Berlin. (Deutsche Fassung von EN 50716:2023; English title: Railway Applications – Requirements for software development)

Ferrari, A., ter Beek, M. H., Mazzanti, F., Basile, D., Fantechi, A., Gnesi, S., ... Trentini, D. (2019). Survey on formal methods and tools in railways: the ASTRail approach. In *International conference on reliability, safety, and security of railway systems* (pp. 226–241). doi: 10.1007/978-3-030-18744-6_15

Friedenberger, D., Pirl, L., Boockmeyer, A., Schmid, R., & Polze, A. (2025). A train dispatcher in the cloud generated from rdf models. In *2025 ieee 22nd international conference on software architecture companion (icsa-c)* (pp. 111–119). doi: 10.1109/ICSA-C65153.2025.00023

Heydari, S. A. (2023). Model-based systems engineering (MBSE) for complex engineering projects. *Management Strategies and Engineering Sciences*, *5*(2), 22–31.

Hillairet, G., Bertrand, F., & Lafaye, J.-Y. (2008, October). Bridging EMF applications and RDF data sources. In *4th international workshop on Semantic Web Enabled Software Engineering (SWESE) at ISWC'08* (p. 26). Karlsruhe, Germany. Retrieved from https://hal.science/hal-00385823

Huldt, T., & Stenius, I. (2019). State-of-practice survey of model-based systems engineering. , *22*(2), 134–145. doi: 10.1002/sys.21466

Jackson, M., Wilkerson, M., & Castet, J.-F. (2016). Exposing hidden parts of the SE process: MBSE patterns and tools for tracking and traceability. In *2016 ieee aerospace conference* (pp. 1–12). doi: 10.1109/AERO.2016.7500594

Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, *5*(11), 558–562. doi: 10.1145/368996.369025

Klare, H., Kramer, M. E., Langhammer, M., Werle, D., Burger, E., & Reussner, R. (2021). Enabling consistency in view-based system development – the Vitruvius approach. *Journal of Systems and Software*, *171*, 110815. doi: 10.1016/j.jss.2020.110815

Meng, B., Paul, S., Varanasi, S. C., Alexander, C., Mertens, E., Moitra, A., ... Durling, M. (2025). RITE: An IDE for assurance and certification of software and systems. In *2025 aiaa datc/ieee 44th digital avionics systems conference (dasc)* (pp. 1–10). doi: 10.1109/DASC66011.2025.11257205

Papke, B., Hause, M., Hetherington, D., McGervey, S., & Rodriguez, S. (2023). MBSE model management pain points–wait, this looks familiar! In *Incose international symposium* (Vol. 33, pp. 273–289). Retrieved from https://www.incose.org/wp-content/uploads/2026/01/Paper-128.pdf

Ramesh, B., & Jarke, M. (2001). Toward reference models for requirements traceability. *IEEE Transactions on Software Engineering*, *27*(1), 58–93. doi: 10.1109/32.895989

Sendall, S., & Kozaczynski, W. (2003). Model transformation: the heart and soul of model-driven software development. *IEEE software*, *20*(5), 42–45. doi: 10.1109/MS.2003.1231150

Technical Operations — International Council on Systems

Engineering (INCOSE). (2007). *Systems engineering vision 2020* (No. INCOSE-TP-2004-004-02). Retrieved from https://sdincose.org/wp-content/uploads/2011/12/SEVision2020_20071003_v2_03.pdf

Walter, B., Kaiser, D., & Rudolph, S. (2019). From manual to machine-executable model-based systems engineering via graph-based design languages. In *Modelsward* (pp. 201–208). doi: 10.5220/0007236702010208

Yang, L., Cormican, K., & Yu, M. (2019). Ontology-based systems engineering: A state-of-the-art review. *Computers in Industry*, *111*, 148–171. doi: 10.1016/j.compind.2019.05.003

Zhang, W., Cockburn, C., Henshaw, M., Douglas, P., Palmer, P., Olivier-Myall, J., & Ji, S. (2025). MBSE co-pilot: A research roadmap. *Systems Engineering*. doi: 10.1002/sys.70011

About the authors

Dirk Friedenberger is a Ph.D. candidate at the Operating Systems and Middleware group of Prof. Andreas Polze at the Hasso Plattner Institute, University of Potsdam. His main research interests include model-based approaches in systems engineering, with a particular focus on the digitalization of railway systems. You can contact the author at dirk.friedenberger@deutschebahn.com.

Jonas Henschel is a Ph.D. candidate at the Operating Systems group of Prof. Matthias Werner at the Department of Computer Science, Technical University of Chemnitz. His main research interests include static analysis of distributed systems focusing on their application to functional and meta-functional properties of Quantum-Key-Distribution networks. You can contact the author at jonas.henschel@informatik.tu-chemnitz.de.

Lukas Pirl is a Ph.D. candidate at the Operating Systems and Middleware group of Prof. Andreas Polze. His main research interest is the field of dependability, especially fault tolerance, experimental assessments through fault injection, dependable operation, and how generic approaches can benefit from domain-specific circumstances. He contributed to multiple publicly funded research projects in the area of the digitalization of railway systems. You can contact the author at lukas.pirl@hpi.de.

Can André Dautel is a Ph.D. candidate at the avionics platform group of Prof. Bjoern Annighoefer. His main research interest focuses on key distribution in self-adaptive and integrated modular avionic platforms. It includes safety analysis and domain-specific modeling. You can contact the author at can.dautel@ils.uni-stuttgart.de.

Andreas Polze is the Operating Systems and Middleware Professor at the Hasso Plattner Institute for Software Engineering at University Potsdam, Germany. He received a doctoral degree from Freie University Berlin in 1994 and a habilitation degree from Humboldt University Berlin in 2001, both in Computer Science. His research focuses on architectures of operating systems, component-based middleware, and predictable distributed and cloud computing. You can contact the author at andreas.polze@hpi.de.