Aalto University
School of Engineering

Master’s Programme in Mechanical Engineering

Formalizing dynamic requirements in simulation-driven collaborative design of powertrains

Haitham Al-Shami

Master’s Thesis
2024

Aalto University
School of Engineering

Author Haitham Al-Shami

Title Formalizing dynamic requirements in simulation-driven collaborative design of powertrains

Degree programme Mechanical Engineering

Major Mechatronics

Supervisor Assistant Professor Raine Viitala

Advisors D.Sc (Tech.) Riku Ala-Laurinaho, M.Sc (Tech.) Miro Eklund

Date 18 November 2024 **Number of pages** 75+16 **Language** English

Abstract

Simulation-driven design is widely used to develop complex engineering systems like powertrains. Engineers implement simulations to analyze component interactions, assess system safety requirements, optimize design efficiency, and refine designs before physical prototyping. A central challenge in this process is verifying that systems meet their specified requirements. However, translating these requirements into a machine-readable, unambiguous format that is clear to all stakeholders—such as component providers and system designers—remains difficult.

This thesis introduces a novel approach to dynamic requirement verification in powertrain system design, utilizing an ontology-based framework that integrates the System Structure and Parameterization (SSP) and Functional Mock-up Interface (FMI) standards. The methodology formalizes dynamic requirements using Metric Temporal Logic (MTL) within an ontological structure to represent time-dependent behaviors and constraints. The implementation involves developing an ontology using Protégé, incorporating existing ontologies like the Vehicle Signal Specification Ontology (VSSo) and the Sensor, Observation, Sample, and Actuator (SOSA) ontology, and creating a parser to generate executable verification commands.

The approach was validated using a ship powertrain case study with a known input drive cycle. The study employed both co-simulation and model exchange Functional Mock-up Units (FMUs) to represent system components, with simulations executed in OMSimulator. A requirement verification tool assessed the compliance of the simulated system behavior against the formalized requirements.

The results demonstrated that the developed ontology accurately captured complex dynamic requirements from expert ship designers; integration with SSP and FMI standards improved system-level interoperability. The verification system precisely detected temporal deviations from specified requirements. This research advances systems engineering by offering a structured approach that enables precise specification of dynamic requirements, thereby improving communication and development efficiency in powertrain engineering.

Keywords Requirement Engineering, System Structure and Parameterization, Functional Mock-up Interface, Requirement Verification, Ontology

Preface

This thesis is the culmination of collaborative research at Aalto University. I would like to express my sincere gratitude to my supervisor, Assistant Professor Raine Viitala, for his guidance and constructive feedback that helped shape and refine this work. I am deeply grateful to my advisor D.Sc (Tech.) Riku Ala-Laurinaho for his constant guidance, continuous support, and feedback throughout the duration of this work. Special thanks to M.Sc (Tech.) Miro Eklund for his expertise in MTL and verification systems, which shaped this work.

I would also like to thank D.Sc (Tech.) Zheng Yuan for sharing his knowledge of ontologies and their design methods. Finally, I want to thank my family for their unwavering support and endless encouragement throughout my studies. Their belief in me has been a constant source of motivation and strength in this journey.

Otaniemi, 18 November 2024

Haitham Al-Shami

Contents

Symbols and abbreviations

τ	Time sequence $(\tau_0, \tau_1, ...)$
ρ	Timed word (sequence of events and timestamps)
σ	Event sequence $(\sigma_0, \sigma_1, ...)$
ϕ, ψ	MTL formulas
$\Diamond_I$	Eventually operator (within interval I)
$\Box_I$	Always operator (within interval I)
$\mathcal{U}_I$	Until operator (within interval I)
$\wedge$	Logical AND
$\vee$	Logical OR
$\neg$	Logical NOT
$\rightarrow$	Implies
$\leftrightarrow$	Equivalence
$\top$	True
$\bot$	False
$\| =$	Satisfaction relation ("satisfies")
I	Time interval [a,b] where a,b are time bounds in $\mathbb{N} \cup \{\infty\}$

Abbreviations

ATP	Automated Theorem Proving
BDD	Binary Decision Diagram
BFO	Basic Formal Ontology
CS	Co-Simulation
DTDL	Digital Twin Definition Language
DT	Digital Twin
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
IoT	Internet of Things
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
LTL	Linear Temporal Logic
MBSE	Model-Based Systems Engineering
ME	Model Exchange
MIL	Model-in-the-Loop
MTL	Metric Temporal Logic
ODE	Ordinary Differential Equation
OEM	Original Equipment Manufacturer
OSLC	Open Services for Lifecycle Collaboration
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	RDF Schema
SCL	Simantics Constraint Language
SHACL	Shapes Constraint Language
SIL	Software-in-the-Loop
SMT	Satisfiability Modulo Theory
SOSA	Sensor, Observation, Sample, and Actuator
SPARQL	SPARQL Protocol and RDF Query Language
SSN	Semantic Sensor Network
SSP	System Structure and Parameterization
SysML	Systems Modeling Language
TLA+	Temporal Logic of Actions Plus
TTL	Turtle (Terse RDF Triple Language)
UML	Unified Modeling Language
VSSo	Vehicle Signal Specification Ontology
WoT	Web of Things
XML	eXtensible Markup Language

1 Introduction

1.1 Background

Systems engineering addresses the challenges of developing complex systems through standardized tools and methods. Modern powertrains exemplify these challenges, incorporating components from multiple manufacturers that must work together seamlessly. Model-Based Systems Engineering (MBSE) has emerged as a structured approach to managing this complexity throughout the system lifecycle, utilizing digital models for information exchange and verification.

Within MBSE, requirements engineering plays an important role in defining system functionality and performance criteria. Requirements serve as benchmarks for system validation, with both system designers and component manufacturers contributing to their definition. Complex systems like powertrains typically involve two types of requirements: static requirements that remain constant, and dynamic requirements that specify time-dependent behaviors. Static requirements can be verified through direct property comparison (e.g., maximum power ratings, physical dimensions). However, dynamic requirements require simulation-based verification to evaluate system behavior over time, such as the response time of a component when a specific condition is met. The verification of dynamic requirements presents particular challenges, as they must capture complex temporal relationships between system components.

Modern powertrain development relies on close collaboration between system designers and Original Equipment Manufacturers (OEMs). OEMs provide component models that system designers must integrate while ensuring that both component-level and system-level requirements are met. This integration of models and requirements has led to the emergence of Modeling and Simulation-Based Systems Engineering [1], where simulation-based verification helps evaluate system behavior before physical implementation. This thesis focuses specifically on formalizing and verifying dynamic requirements within MBSE, addressing the gap between requirement specification and verification in complex system development.

1.2 Research Problem

In collaborative design environments for complex systems like ship powertrains, challenges remain in effectively communicating and standardizing requirements across different stakeholders. The Functional Mock-up Interface (FMI) [2] and System Structure and Parameterization (SSP) [3] standards are tools for model sharing and co-simulation that are becoming increasingly popular. FMI is used for the creation of Functional Mock-up Units (FMUs), which represent standardized component models often provided by Original Equipment Manufacturers (OEMs). The SSP standard offers a framework for system designers to integrate these components into a cohesive system model.

Despite adopting these standards in sectors such as ship digital twins [4], aeronautics [5], and the automotive industry [6], a significant gap remains in requirement representation and communication. Even seemingly-straightforward requirements like "motor torque must not exceed 1000 Nm during normal operation" can lead to misinterpretation — terms like "normal operation" may hold different meanings for component providers and system integrators. Converting natural language requirements into formal, machine-readable formats, poses multiple challenges: temporal dependencies must be precisely specified, relationships between different requirements must be explicitly defined, and ambiguous terms must be replaced with standardized definitions.

The lack of standardized requirement representation leads to misalignment between component-level specifications and system-level requirements. This misalignment impairs validation of system performance against design expectations and restricts co-simulation capabilities, particularly in verifying both static and dynamic requirements throughout the system lifecycle.

This thesis addresses these challenges by developing an ontology-based approach that standardizes requirement representation and communication in collaborative design environments. The research examines the following questions:

1. How can an ontology-based approach standardize system architecture and requirements across different sources while ensuring consistent integration of static and dynamic specifications?

2. How can the variables defined by component providers be effectively mapped and linked to a standardized requirement interpretation system?

3. How can simulation-based verification ensure compliance with formalized requirements within this ontological structure?

Addressing these questions advances systems engineering and improves collaborative design processes. The proposed ontology-based approach establishes a standardized method for requirement representation and communication, bridging the gap between component specifications and system-level integration.

1.3 Objectives

The primary objective of this thesis is to develop an ontology that standardizes dynamic requirement definitions for powertrain components and systems. This ontology provides a consistent language for expressing requirements that bridges the gap between component manufacturers and system integrators.

The second objective is to develop a parsing mechanism that converts ontological requirements into machine-executable verification commands using Metric Temporal Logic (MTL) - a formal language specifically designed for specifying time-bounded properties. This conversion enables systematic requirement verification through simulation-based testing.

The third objective validates the approach through a ship powertrain case study. The study implements the ontology using System Structure and Parameterization (SSP) for system representation and Functional Mock-up Units (FMUs) for component modeling. The parser transforms ontological requirements into executable commands for requirement verification during simulated operations.

These three objectives are shown in Figure 1. This research advances Model-Based Systems Engineering by providing a structured method for requirement definition and verification in complex, collaborative design environments.

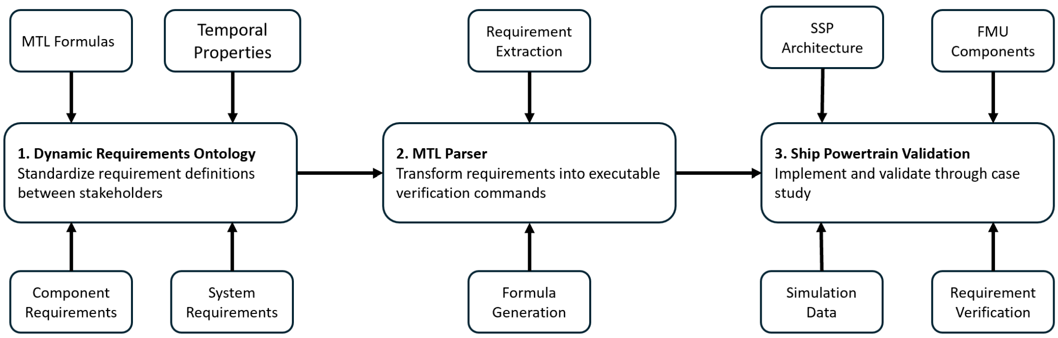

Figure 1: Overview of thesis objectives and their components. The progression shows three main stages: (1) developing an ontology for standardizing requirements, (2) creating an MTL parser for requirement transformation, and (3) validating the approach through a ship powertrain case study. Each stage shows its input and output elements.

1.4 Scope

This research focuses on defining requirements in a standardized format for powertrains, although the developed method may extend to other domains such as software systems or different engineering fields. These potential applications remain outside this thesis's validation scope and could be considered for future work. The implementation employs SSP and FMI standards exclusively, chosen for their widespread tool support and established use in the field [7]; further integration with other engineering standards remains as future work. The research scope includes only requirements that can be formalized in Metric Temporal Logic and verified through simulation. This focus enables systematic verification of dynamic requirements through ontological representation while acknowledging that certain requirements may require different verification approaches.

1.5 Methods

This research follows four main methodological approaches. The first phase employs systematic literature review methodology, using established databases (IEEE Xplore, Scopus, Web of Science) with specific search criteria and inclusion/exclusion rules to analyze current research in requirement verification and ontology-based systems.

The second phase employs Ontology Engineering (OE) principles, incorporating established design patterns and evaluation techniques. This includes the use of competency questions for ontology validation and automated reasoning for consistency checking. The third phase applies formal methods through Metric Temporal Logic (MTL), leveraging term rewriting techniques for requirement verification. The approach combines semantic technologies with simulation-based verification methods.

The final phase uses experimental validation through simulation, employing both co-simulation and model exchange FMUs. The validation process includes comparative analysis of different simulation approaches and systematic verification of requirements through verification algorithms. Figure 2 illustrates the parallel workflows of system development: system designers using SSP standard for system architecture and integration, while component manufacturers provide validated FMU components. Together, these components form a structured development process for collaboration between stakeholders.

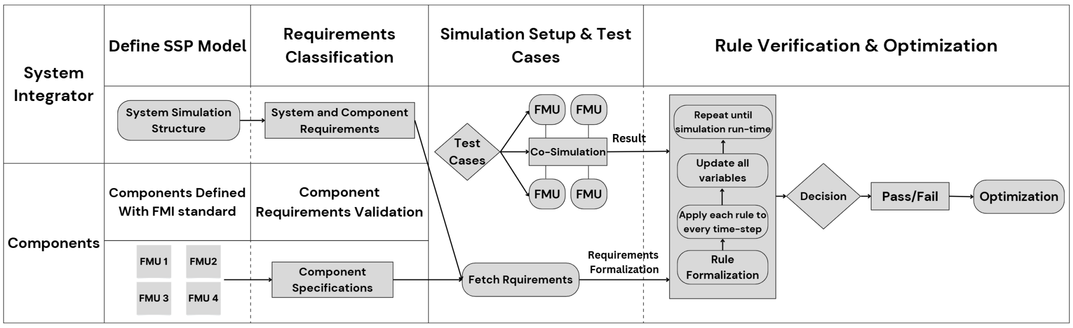

Figure 2: Workflow overview of requirement verification in simulation-driven design. The process flows from system definition (left) through requirements formalization and simulation to verification and optimization (right). The diagram shows the parallel workflows of system integrators and component manufacturers.

2 Research Background

This chapter presents the concepts needed to understand requirement verification in powertrain design. Starting with systems engineering in Section 2.1, we establish the foundation for managing complex systems and their requirements. Digital twins and simulation, discussed in Section 2.2, demonstrate how virtual models enable system testing before physical implementation. The SSP and FMI standards, detailed in Section 2.3, provide the technical infrastructure for sharing and connecting these models. Section 2.4 introduces ontological approaches that structure requirements in an unambiguous, machine-readable manner. Finally, Section 2.5 examines formal verification methods for validating these requirements. Together, these foundations establish a comprehensive framework for formalizing dynamic requirements in collaborative design.

2.1 Systems Engineering

Systems Engineering (SE) provides a methodological framework for designing and managing complex systems throughout their lifecycles [8]. SE addresses a fundamental challenge in complex systems: components that function correctly in isolation may fail when integrated. This challenge is particularly relevant in powertrain design, where multiple interdependent components must work together seamlessly. Within SE, requirement engineering plays a central role, establishing clear specifications for system functionality and performance. Understanding SE principles is therefore essential for implementing effective requirement engineering and verification methods, which form the core focus of this thesis.

2.1.1 Model-Based Systems Engineering (MBSE)

Model-Based Systems Engineering (MBSE) represents a shift from document-centric to model-centric approaches in systems engineering (SE). The International Council on Systems Engineering (INCOSE) defines MBSE as "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" [9]. MBSE relies on several principles:

1. **Standardized modeling languages:** MBSE employs languages like Systems Modeling Language (SysML) and Unified Modeling Language (UML) to create consistent system representations. SysML dominates systems engineering, however, it struggles to represent dynamic requirements and formal semantics. Powertrains and other complex systems require extending SysML with additional formalisms like ontologies for comprehensive requirement verification [10].

2. **Domain integration:** MBSE unifies models across engineering disciplines. This unification enables mechanical, electrical, and software components to interact seamlessly in complex systems like powertrains [11].

3. **Requirement traceability:** MBSE maintains explicit links between requirements, design elements, and verification procedures. This linking ensures each system component maps to specific requirements.

MBSE improves engineering design through virtual prototyping, continuous requirement verification, and systematic design space exploration, allowing engineers to evaluate multiple design alternatives before physical implementation. Throughout development, MBSE leverages simulation to validate system designs, enabling comprehensive system testing. However, practical challenges persist, such as integrating legacy systems and maintaining compatibility between various engineering tools [12]. Figure 3 shows the main components of an MBSE approach for powertrain systems.

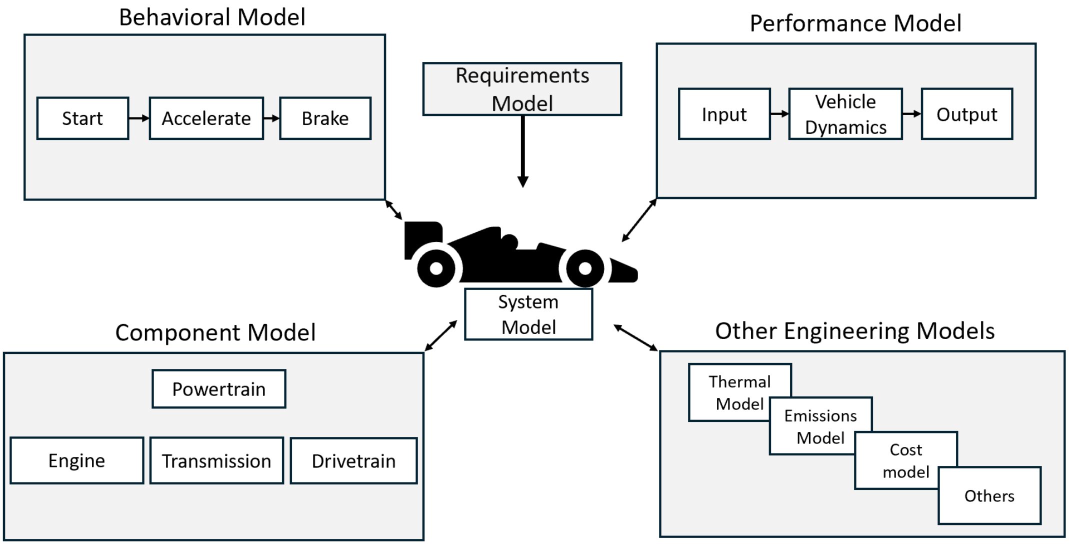

Figure 3: Representation of a simplified MBSE diagram focused on powertrain systems.

Among MBSE's components, requirement engineering is central to this thesis. Modern systems' complexity makes formalizing and verifying dynamic requirements challenging. The next section examines requirement engineering and its role in managing system specifications throughout development.

2.1.2 Requirement Engineering

Requirement engineering (RE) defines the characteristics, goals, and expected outcomes of a system [13]. RE involves multiple stages: understanding the domain, gathering information, defining needs, evaluating proposals, resolving conflicts, recording decisions, and adapting requirements over time. Effective RE practices address contextual goals (why), functionalities (what), and constraints (how) of a system [14]. Requirements can be categorized into multiple general types:

1. Functional Requirements: These specify what the system should do.

2. Non-functional Requirements: Determine the requirements of how the system should perform its functions, including reliability, usability, etc.

3. Domain Requirements: Specific to the application domain (e.g., marine engineering for powertrains).

4. User Requirements: High-level statements of user expectations.

5. System Requirements: Detailed descriptions of system behavior.

6. Interface Requirements: Specify system interactions with external entities.

7. Constraint Requirements: Define limitations within which the system must operate.

Requirements must be formalized using well-defined syntax and semantics, often employing a combination of natural language and formal notations [15]. The IEEE 830-1998 [16] standard provides guidelines for Software Requirements Specifications (SRS), stipulating that requirements should be unambiguous, verifiable, consistent, modifiable, and traceable. Requirements should be organized hierarchically, as illustrated in Figure 4.

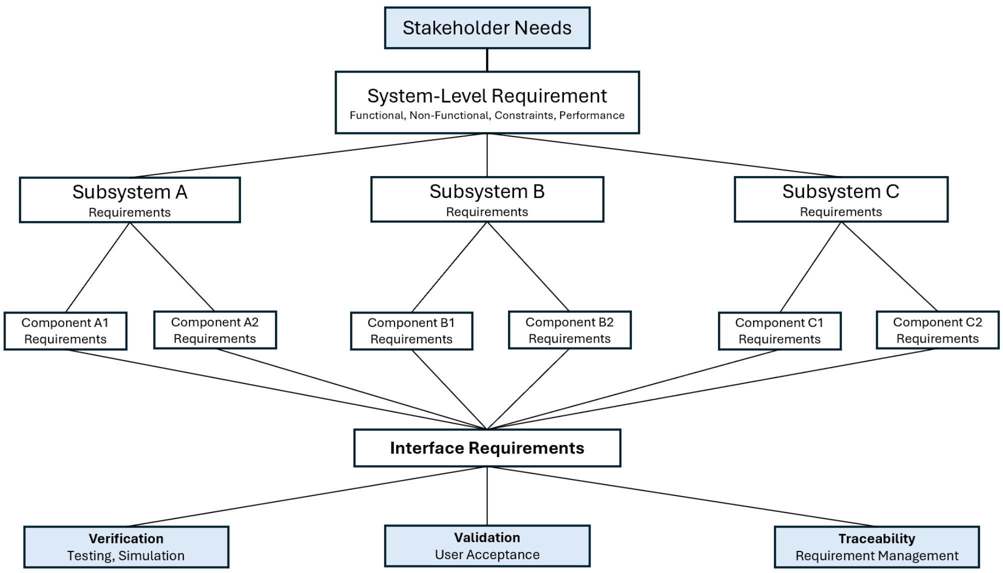

Figure 4: Hierarchy of requirements as they are defined within a system. The interface requirements link the internal system requirements to external methods imposed on them.

Traceability links requirements to system elements. MBSE tools support SysML's <trace> relationships between model elements [17]. For complex systems like powertrains, traceability analysis helps maintain consistency across large-scale requirement sets [18]. An example of traceability is as follows:

1. System-Level Requirement (SYS-REQ-001): "The vehicle accelerates from 0 to 100 km/h in less than 8 seconds under standard test conditions."

2. Subsystem-Level Requirement (PWT-REQ-001): "The powertrain subsystem shall provide a peak power output of at least 200 kW."

3. Component-Level Requirement (ENG-REQ-001): "The engine shall deliver a maximum torque of 350 Nm at 4000 RPM."

This example demonstrates a traceability path from component to system level, which is essential for system validation. Traditional systems engineering approaches, such as SysML and UML, have proven effective for defining static requirements and documenting system architecture [10]. However, the increasing complexity of modern systems, particularly in powertrain design, demands more advanced methodologies for handling dynamic requirements and complex component interactions.

The concept of digital twins, with its continuous data exchange between physical and virtual systems, offers a new paradigm for addressing these challenges. Integrating systems engineering with digital twins enables comprehensive requirement management, verification, and validation throughout the system lifecycle.

2.2 Digital Twins in System Simulation

In the context of this thesis, digital twins are primarily used to make models available and discoverable as Functional Mock-up Units (FMUs). Co-simulation allows different component models, often developed separately, to be simulated together. Digital twins aid in this by providing a unified virtual environment where these diverse components can interact. In collaborative design, digital twins offer a shared platform for system designers and component manufacturers to test and optimize their designs without physical prototypes.

2.2.1 Concept of Digital Twins

Digital twins are digital replicas of physical systems. Grieves and Vickers [19] define a digital twin as "a set of virtual information constructs that fully describes a potential or actual physical manufactured product from the micro atomic level to the macro geometrical level." These virtual representations enable simulation, verification, and optimization of complex systems composed of multiple components.

Digital twins may exist at different stages of development, including digital twin prototypes that typically precede physical implementation. In this thesis, we work with such prototypes, where only the virtual models exist. A digital twin system generally consists of three main elements: the physical asset, its digital counterpart, and the bidirectional data flow between them, as shown in Figure 5. Sensors and user inputs typically collect data from the physical asset, which can then be transmitted to the virtual space. This digital model may support various functions such as diagnostics, forecasting, and optimization, potentially creating a closed-loop system for continuous improvement and real-time decision-making [20].

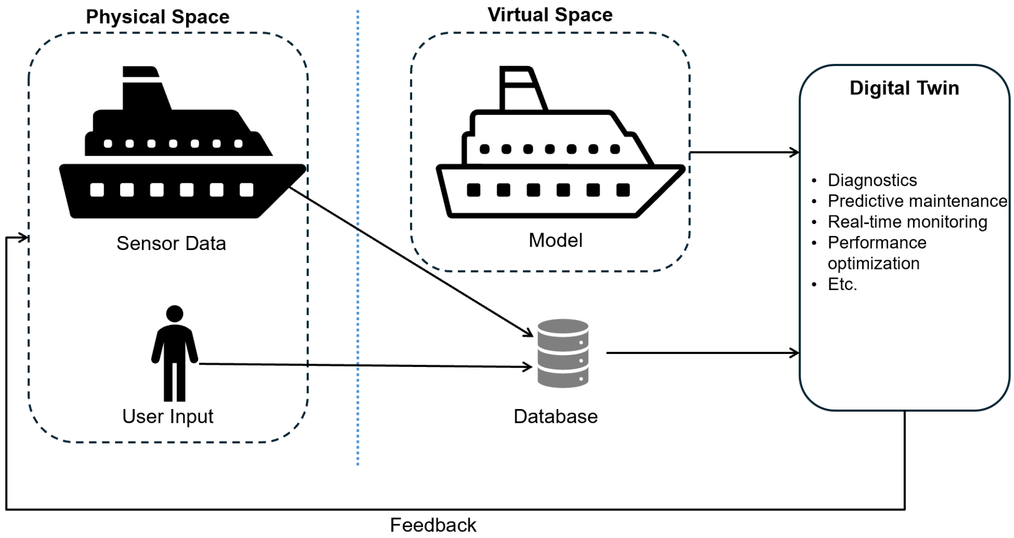

Figure 5: Digital twin system with continuous data exchange between physical assets and their virtual representations.

Despite their potential, digital twins face adoption challenges. Sharma et al. [21] identified limitations in universal frameworks, evolving standards, and dependencies on emerging technologies like IoT and machine learning. In powertrain design, these manifest as integration challenges between component models and inconsistent requirement representations across stakeholders. This lack of standardization hinders collaborative design environments by impeding precise specification exchange. These challenges motivate this thesis's focus on developing standardized methods for requirement representation and verification in powertrain systems.

2.2.2 Digital Twin Standards

To address these adoption challenges, various standardization efforts for Digital Twins (DTs) have emerged across industries. The wide range of applications and requirements makes standardization difficult. In the context of Industry 4.0, where manufacturing systems are becoming increasingly digital and interconnected, Jacoby and Uslánder [22] identify interoperability between systems with varying standards and protocols as a barrier to DT implementation.

Several initiatives aim to create standardized frameworks for digital twins. Microsoft developed the digital twin Definition Language (DTDL) for use within their Azure services, providing a framework to define properties, telemetry, commands, and relationships of digital twins [22]. The Asset Administration Shell (AAS), developed by Plattform Industrie 4.0, serves as a digital representation of assets, covering operational and lifecycle management information [23].

The Web of Things (WoT) group proposed the Thing Description (TD) ontology, which González-Usach et al. [24] extended to create the WoT Digital Twin (WoTDT) ontology. WoTDT provides formal semantics to represent five dimensions of digital twins for semantic discovery and a standardized approach to describe digital twin capabilities.

Digital twin implementations often face compatibility issues due to varied data formats across systems. Jacoby and Usländer [22] compared DTDL and AAS, finding that both support multiple data formats including XML, JSON, JSON-LD, and RDF. This multi-format support addresses interoperability challenges between different platforms. Similarly, the WoTDT ontology was developed using Ontology Engineering (OE) practices to ensure quality and broad applicability.

To illustrate the application of DTDL, consider the following example of DTDL syntax used to define a thermostat:

```
{
  "@id": "dtmi:example:Thermostat;1",
  "@type": "Interface",
  "displayName": "Thermostat",
  "contents": [
    {
      "@type": "Telemetry",
      "name": "temperature",
      "schema": "double"
    },
    {
      "@type": "Property",
      "name": "targetTemperature",
      "schema": "double",
      "writable": true
    },
    {
      "@type": "Command",
      "name": "reset"
    }
  ]
}
```

Listing 1: Example definition of a thermostat DT using DTDL. This DTDL example demonstrates how digital twin capabilities and properties are structured in a machine-readable format.

Despite standardization efforts, challenges persist. Stadtmann et al. [25] identified barriers in the wind energy sector including data quality limitations, incomplete standards, and security concerns. These challenges extend across industries, particularly in integrating legacy systems with modern digital twin technologies. Vuković et al. [26] addressed this through a layered architecture approach, enabling integration of older systems while managing data format incompatibilities and real-time exchange requirements.

Recent implementations demonstrate progress in practical applications. Cavalieri and Gambadoro [27] enhanced interoperability by mapping DTDL to OPC UA, while O'Connell et al. [28] showed how digital twin standards can create interconnected systems. Applications range from predictive maintenance in manufacturing [29] to autonomous vehicle testing [30] and AI-driven system optimization [31].

2.2.3 Integration of Systems Engineering and Digital Twins

Systems engineering provides a structured framework for complex system development, while digital twins offer dynamic virtual representations of physical assets with continuous data updates. In MBSE, formal modeling languages like SysML link directly to digital twin implementations, enabling real-time validation of system behavior against requirements. For example, in powertrain development, an MBSE model defining system architecture couples with a digital twin representation to enable continuous system verification [32].

However, integrating MBSE tools with digital twin systems presents challenges. Different data storage and communication methods between tools create interoperability issues [33]. Large systems generate vast amounts of data, requiring cloud platforms and distributed computing architectures for effective management [34]. These challenges have driven the development of specialized standards for simulation and model exchange.

The System Structure and Parameterization (SSP) and Functional Mock-up Interface (FMI) standards address these integration challenges. These standards enable creation and exchange of virtual system representations through standardized interfaces. In powertrain design, where multiple components must interact seamlessly, SSP and FMI allow collaborative development through structured model exchange and co-simulation. The following section examines these standards in detail.

2.3 Standards in Co-Simulation

The System Structure and Parameterization (SSP) and Functional Mock-up Interface (FMI) standards integrates varied technical viewpoints in system simulation [35]. The FMI standard focuses on model exchange through Functional Mock-up Units (FMUs), whereas the SSP standard provides the framework to connect these models in a unified simulation environment.

2.3.1 SSP Standard Components

SSP uses a tool-independent XML schema to define system components, their interconnections, and parameter settings. As shown in Figure 6, an SSP package consists of four main components:

1. System Structure Description (SSD): Describes the system's hierarchical structure, components, connections and overall layout, with parameters definable at both system and component levels.

2. System Structure Buses (SSB): Manages signal definitions within the system, specifying data types (e.g., real, integer, boolean) and standardizing signal connections between components.

3. System Structure Parameter Values (SSV): Defines and stores parameter values across components and simulations.

4. System Structure Parameter Mapping (SSM): Links source parameters to system or component parameters, supporting transformations where needed.

SSP packages these components in a ZIP archive, supporting multiple system versions in a single portable unit. While FMI handles individual model exchange, SSP provides the framework for connecting multiple models in a unified simulation environment [3], allowing system configuration management and collaborative design.

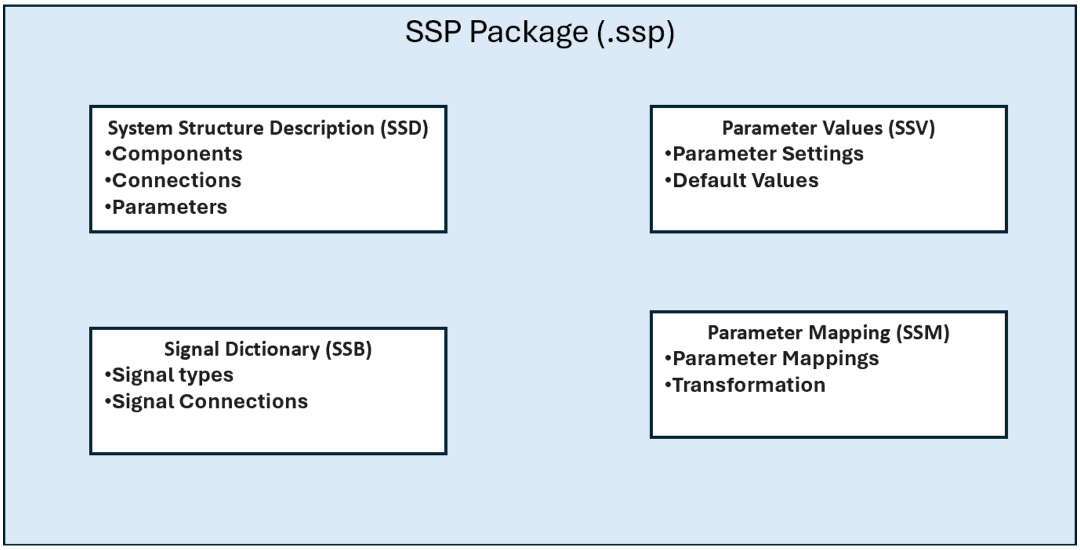

Figure 6: The four main files required in an SSP package: SSD, SSB, SSV, and SSM.

The practicality of the SSP standard has been demonstrated through various implementations. SSP enables FMU network exchanges across different simulation platforms, including Model-in-the-Loop (MIL), Software-in-the-Loop (SIL), and Hardware-in-the-Loop (HIL). Kohler et al. [36] validated SSP's capabilities through three complementary tools. The Integration Tool (Model.CONNECT™) demonstrated system setup management and visualization capabilities, while their Co-Simulation Browser supported collaborative work and mobile simulation execution. Their FMI Bench tool completed the validation by successfully testing SSP-FMU integration through an electric drive model.

SSP uses XML for system structure description and ZIP for package compression, allowing straightforward information exchange between software tools. The standard's support for multiple system versions within a single package through the SSD file lets engineers compare design alternatives easily. While SSP provides the framework for system integration, it relies on standardized component models for simulation - this is where the Functional Mock-up Interface (FMI) becomes useful.

2.3.2 FMI Standard and Simulation

The Functional Mock-up Interface (FMI) is an open standard for exchanging dynamic models in simulation environments. Developed by the Modelica Association Project, FMI uses XML files, binaries, and C code, supported by over 170 tools [2].

FMI specifies three interfaces: Model Exchange (ME), Co-Simulation (CS), and Scheduled Execution (SE). ME exposes ODEs (Ordinary Differential Equations) to external solvers, CS couples simulation tools with internal solvers, and SE activates model partitions via external schedulers for real-time platforms. Figure 7 illustrates the main differences between ME and CS FMUs.

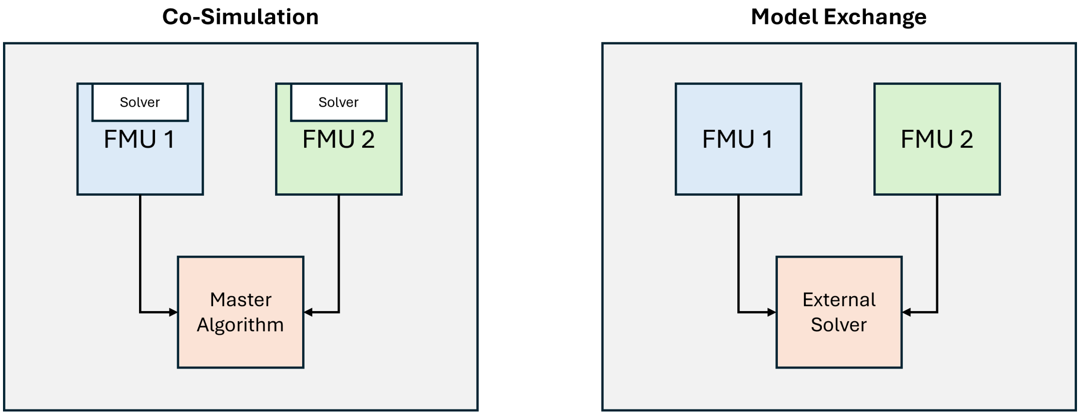

Figure 7: Co-simulation with separate solvers vs. model exchange with an external solver.

FMI 3.0 expands the standard's capabilities with virtual electronic control units (vECUs), event timing clocks, new data types, and array handling. Gomes et al. [37] demonstrated these advances through Synchronous Clocked Simulation and Scheduled Execution interfaces in control systems with varied task rates. This thesis uses FMI 2.0 for dynamic requirement verification in co-simulation, as FMI 3.0's recent release means limited adoption in current simulation software. Table 1 compares the features across FMI versions.

Feature	FMI 1.0	FMI 2.0	FMI 3.0
Release Year	2010	2014	2021
Model Exchange	Yes	Yes	Yes
Co-Simulation	Yes	Yes	Yes
Scheduled Execution	No	No	Yes
Binary Data Type	No	No	Yes
Array Variables	Limited	Improved	Full support
Clocks	No	No	Yes
Intermediary Update	No	Yes	Enhanced
Backward Compatibility	N/A	With 1.0	With 2.0
XML Schema	Separate ME and CS	Unified	Further unified
Terminal Variables	No	No	Yes
Layered Standard	No	No	Yes

Table 1: Comparison of FMI Versions. FMI 3.0 is a major version update from 2.0, introducing non-backward compatible changes, some backward compatibility may be maintained through a layered standard mechanism. FMUs developed for FMI 2.0 may require modifications to work with FMI 3.0, particularly if they depend on changed XML schemas or specific 2.0 features. [2]

The implementation of FMI presents several challenges. Different modeling tools may interpret the standard inconsistently [38], while co-simulation can introduce numerical instabilities when coupling stiff systems or those with different time constants [39]. Large-scale systems with many coupled FMUs face a trade-off between simulation accuracy and computational cost. Organizations must also balance model sharing and intellectual property protection [40], particularly with black-box models where error handling and diagnostics become complex. Having examined both SSP and FMI standards separately, understanding their integration is needed for implementing complex system simulations.

2.3.3 Integrating SSP and FMU

The integration of System Structure and Parameterization (SSP) and Functional Mock-up Units (FMUs) creates a framework for complex system simulation. SSP defines the system's hierarchical structure, components, connections, and parameter settings, while FMUs handle the dynamic models describing component behavior. This integration allows for the management of complex system interactions and dependencies.

Figure 8 shows how SSP packages integrate with FMUs. The System Structure Description (SSD) acts as a blueprint, defining FMU inputs, outputs, connections, and global parameters. This enables the assembly of complex simulations from pre-existing components.

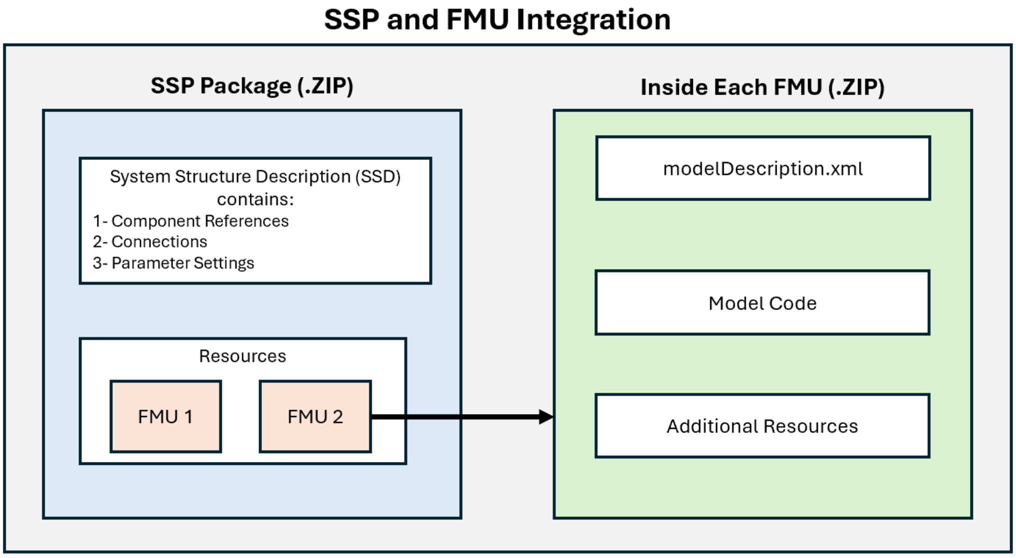

Figure 8: The SSP package contains the system specific files, it also can contain the actual FMUs as resources.

This integration supports modular system design through independent component development and testing before system assembly. SSP's Parameter Sets and Parameter Mapping features allow multiple system configurations without structural changes, streamlining the evaluation of design variants and operating conditions [41].

OMSimulator demonstrates practical SSP-FMU integration as an open-source tool in the OpenModelica suite [42]. The tool supports both model exchange and co-simulation of FMUs, while using SSP for model composition and parameter management. Available as a standalone application, OMSimulator enables simulation of models from multiple sources without Modelica-specific dependencies. Addressing requirement specification and verification requires additional approaches. One such approach is through ontologies, examined in Section 2.4 below.

2.4 Ontology and Requirement Modeling

Requirements form a cornerstone of systems engineering, defining system functionality and performance criteria across all development stages. Despite advances in collaborative design and co-simulation through digital twins, SSP, and FMI, the challenge of communicating requirements precisely across different stakeholders persists. Ontologies address this challenge by providing a formal framework for requirement representation and verification in complex engineering systems. Due to their ease of re-use and extensions, they have been applied to many fields [43].

Ontologies structure domain knowledge through precise semantic relationships, enabling formal requirement representation. Hu et al. [44] and Kaiya and Saeki [45] demonstrated this by developing complementary approaches. Their work includes decomposing requirements into atomic elements with verifiable inference rules, and using directed graphs with first-order predicate logic for requirement verification. These ontology-based methods offer a systematic framework that complements simulation standards [46]. Since this thesis addresses dynamic requirement formalization in powertrain systems, understanding ontology fundamentals becomes central. The following section examines the design principles for effective ontological design.

2.4.1 Ontology Design Principles

Gangemi and Presutti [47] outline practices for robust and user-friendly ontologies, with modular design at the core of complexity management. This approach divides ontologies into smaller, reusable components. Designers can then focus on specific domain aspects, simplifying updates and maintenance. OWL supports this modularity through import statements, enabling ontologies to incorporate content from other sources.

Documentation and annotations form another core principle of ontology design. OWL provides annotation properties such as rdfs:comment and rdfs:label to describe classes, properties, and individuals. These annotations help users understand the ontology's purpose. The reuse of existing ontologies through OWL's import mechanism aligns new ontologies with established standards like FOAF and Dublin Core [47]. Bravo et al. [48] describe a well-defined ontology development procedure consisting of three main steps:

1. Terms definition
2. Identifying ontology components
3. Creating and formalizing individual ontologies using Domain Language (DL) notation

In the terms definition phase, similar terms are grouped into clusters, each representing a distinct domain aspect - belonging to the same class. All components are then identified, ontologies must be designed with the idea of components relating to each other. The final phase involves defining elementary term axioms for each ontology using DL. This establishes hierarchical relationships, specifies data properties (e.g., a person having a name, age, weight), and defines object properties linking concepts within the ontology. Axiomatization sets minimal characteristics for class membership. These include class, cover, closure, data, and object property axioms. Bravo et al. [48] demonstrate these principles in their ontology of medical diseases, which represents disease taxonomy according to the international classification of diseases (ICD-10). Their work details class hierarchies and properties for accurate knowledge representation and reasoning in the medical domain.

Boyce et al. [49] categorize ontologies into three main types based on their scope and application: generic ontologies that describe broad concepts applicable across multiple domains, domain ontologies that focus on well-defined concepts within a specific field, and application ontologies that address particular tasks or applications. Figure 9 illustrates this hierarchy.

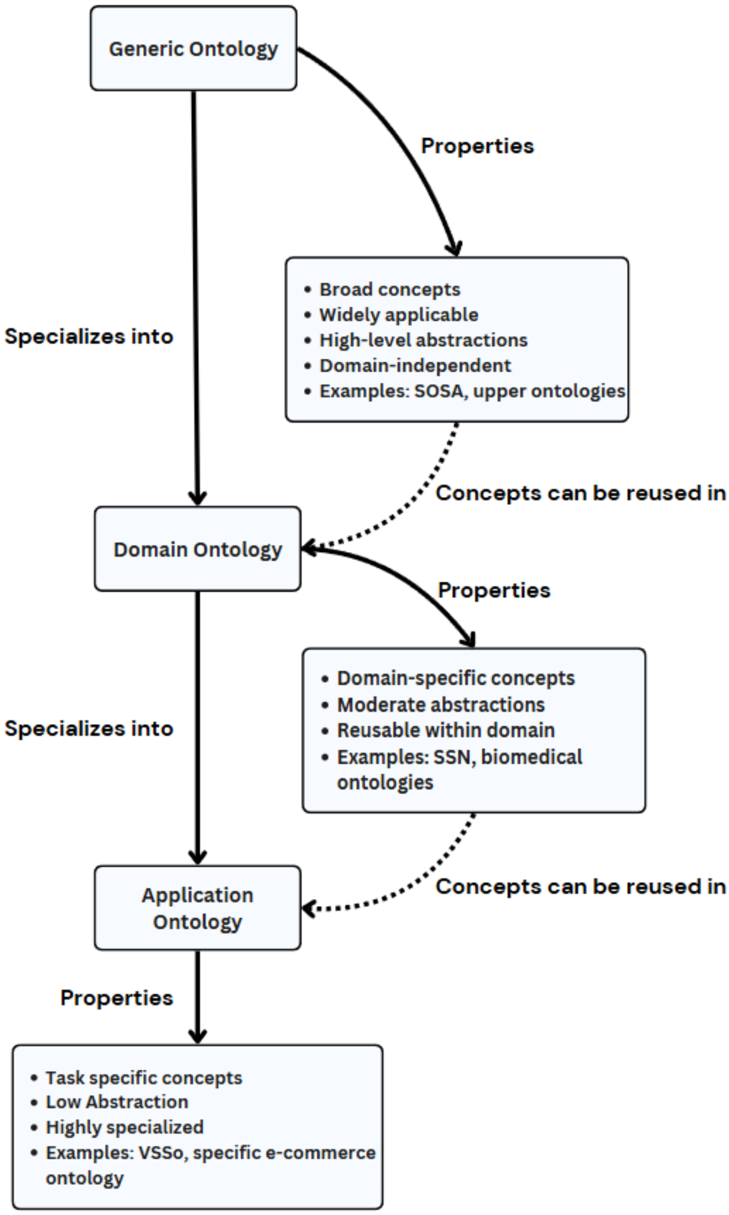

Figure 9: Generic, domain, and application ontologies and their relationships.

Generic ontologies provide widely applicable concepts that serve as a foundation for more specialized ontologies. Domain ontologies, such as the Semantic Sensor Network (SSN), build upon this foundation to specialize in specific fields while maintaining reusability. Application ontologies, at the most specialized level, are highly focused but have limited reusability, as noted by Achichi et al. [50].

Upper-level ontologies extend this hierarchy by representing an even higher level of abstraction, describing general concepts across all knowledge domains. Examples include the Basic Formal Ontology (BFO) and the Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE). Borgo et al. [51] explain that DOLCE, for instance, applies to various domains through four fundamental categories: Endurant, Perdurant, Quality, and Abstract. These upper-level ontologies provide a conceptual framework for more specific ontologies.

Pinto and Martins [52] define the development and maintenance of these ontology types as Ontology Engineering (OE). This process follows a structured lifecycle, as illustrated in Figure 10.

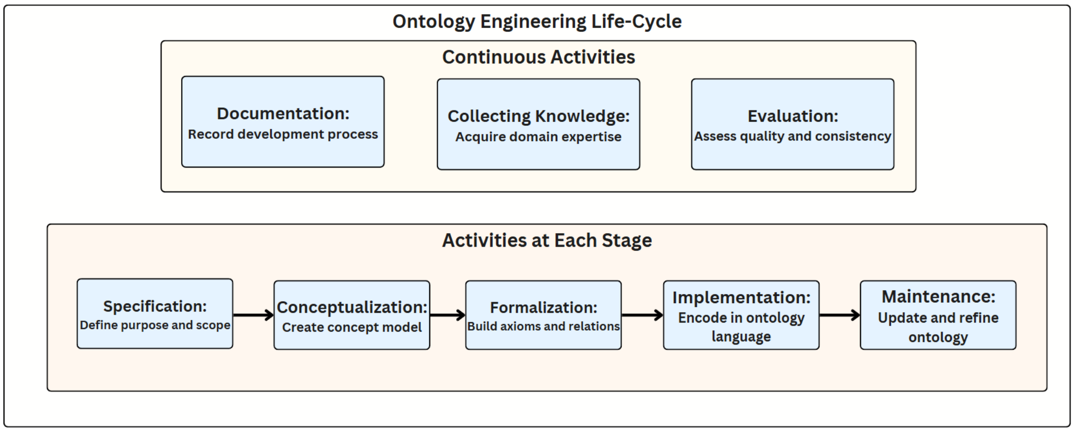

Figure 10: Development stages of an ontology as defined by Pinto and Martins [52].

The ontology development process follows four main stages: Specification defines the ontology's purpose and scope. Conceptualization creates the concept model and establishes relationships. Formalization constructs axioms and defines terms and relations. Implementation and Maintenance realize the ontology in a suitable language and maintain it over time. Continuous documentation and evaluation throughout these stages ensure proper ontology evolution.

Ontology evaluation ensures quality throughout the formalization of complex requirements through three interconnected approaches. The first approach, logical evaluation, employs reasoning engines like Pellet and HermiT to detect contradictions and verify inference paths [53]. The second, data-driven evaluation, relies on competency questions (CQs) to verify that the ontology accurately represents real-world knowledge. These CQs, developed collaboratively with domain experts, serve as functional benchmarks throughout the ontology's lifecycle. The third approach, user-centric evaluation, complements the technical validation by examining practical applicability through expert assessment.

The NeOn methodology provides a structured approach to ontology development through Initiation, Reuse, Design, Implementation, and Maintenance phases [54]. This methodology emphasizes ontology reuse, as demonstrated by Cao et al. [55] in manufacturing domains. Validation employs a task-based approach, using practical data to answer competency questions and demonstrate utility through rule set triggering [56].

Ontologies provide a structured framework for requirement representation and verification [57], particularly suitable for complex engineering systems where requirements must be unambiguous and machine-readable. Their application in requirement formalization forms a critical foundation for this thesis's core contribution: the representation of dynamic requirements. The following section examines how ontologies enable precise requirement formalization in engineering systems.

2.4.2 Requirement Formalization and Ontology Standards

Ontologies naturally lend themselves to requirement formalization through their ability to transform ambiguous natural language into precise, machine-readable statements. Chen et al. [58] demonstrated this capability by developing a method that converts system requirements into explicit ontological rules. For example, a natural language requirement like 'the system should contain exactly four detectors' becomes a formal logical statement:

```
TrafficLightSystem(?tls) Detector(?d1) Detector(?d2) Detector(?d3)
Detector(?d4)
hasComponent(?tls, ?d1) hasComponent(?tls, ?d2) hasComponent(?tls, ?d3)
hasComponent(?tls, ?d4) hasDesignProblem(?tls, noDetectorAmountProblem).
```

This formalization extends to model-based approaches. Chen et al. [58] developed rules to automatically convert Systems Modeling Language (SysML), an extension of the widely used Unified Modeling Language (UML), into ontological representations for verification through reasoning capabilities. Yuan and Tripathi [59] further advanced the field by combining multiple domain ontologies. However, these traditional modeling languages show limitations for complex systems. Torsleff et al. [60] noted their inadequacy in representing dynamic contexts and domain-specific details, necessitating more sophisticated approaches through standardized ontology languages. To address these limitations, several established standards enable formal requirement representation. The Resource Description Framework (RDF) provides a foundation through subject-predicate-object triples that express relationships [61]:

```
<http://example.org/person/jim> <http://example.org/vocab/name> "Jim Doe" .
<http://example.org/person/jim> <http://example.org/vocab/age> "30"^^<http://www.w3.org
    /2001/XMLSchema#integer> .
```

The Web Ontology Language (OWL) extends RDF by adding vocabulary for class hierarchies, property characteristics (such as transitivity or symmetry), and logical constraints between classes [62]:

```
<owl:Class rdf:about="http://example.org/ontology/Person">
  <rdfs:subClassOf rdf:resource="http://example.org/ontology/LivingBeing"/>
</owl:Class>

<owl:ObjectProperty rdf:about="http://example.org/ontology/hasParent">
  <rdfs:domain rdf:resource="http://example.org/ontology/Person"/>
  <rdfs:range rdf:resource="http://example.org/ontology/Person"/>
</owl:ObjectProperty>
```

RDF Schema (RDFS) extends RDF with vocabulary for class hierarchies and properties, while SPARQL enables querying of the resulting ontologies [63]. These standards transform conceptual relationships into implementable frameworks for requirement formalization in systems engineering.

2.5 Requirement Verification

Requirements formalized in ontologies must be verified against actual system behavior to ensure compliance. Formal verification provides this capability by mathematically proving system correctness through various approaches [64]. Two prominent methods are Automated Theorem Proving (ATP) and Model Checking. ATP handles complex component verification efficiently without state space explosion, though its implementation requires significant expertise [65]. Model checking systematically verifies system properties expressed in temporal logic by exploring all possible states and transitions, though it faces scalability challenges due to state space explosion, which recent advances in symbolic model checking help mitigate [66]. For systems with multiple representations, Equivalence Checking uses Binary Decision Diagrams (BDDs) and graph hashing to verify consistency between high-level designs and low-level implementations, managing complexity through system partitioning and variable resubstitution [67].

Simulation-based verification dominates the powertrain industry due to its practical advantages in early design stages. This approach leverages existing simulation tools, computational resources, and established workflows to validate system behavior [68]. For powertrain systems, simulation enables verification of specific scenarios, such as sudden engine power loss or varying load conditions, before physical prototyping. The availability of commercial simulation tools and their integration with design software makes this approach particularly attractive to industry practitioners. Formal verification methods, such as Model Checking, can complement these simulation-based approaches by providing mathematical proofs of system behavior. Figure 11 illustrates the relationship between different verification approaches.

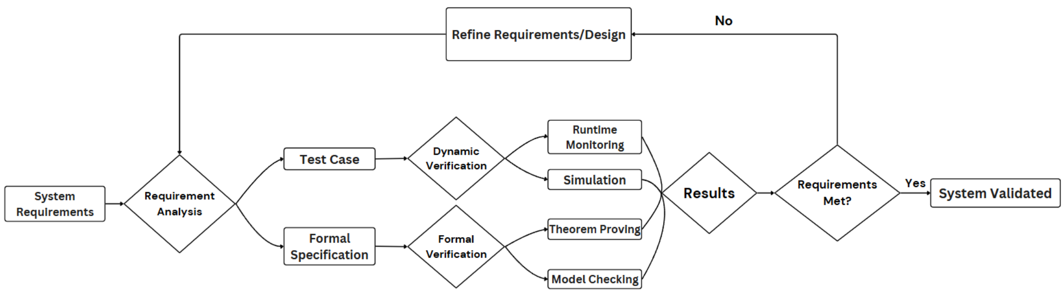

Figure 11: Verification methodology incorporating simulation-based approaches (runtime verification, simulation) and formal methods (theorem proving, model checking) for system validation.

Given the industry's widespread adoption of simulation-based verification, this thesis focuses on this approach while incorporating elements of formal methods. Ontologies play a role in this integration by providing a structured framework for requirement representation. Their ability to formalize domain concepts makes them particularly suitable for representing logical expressions that can be verified through simulation. The following section examines the basics of how ontologies can capture and structure logical formulas for requirement verification.

2.5.1 Formal Requirements

Formal requirements employ mathematical notations to provide precise system specifications, addressing the ambiguity inherent in natural language descriptions [69]. These methods range from Controlled Natural Languages (CNLs) to mathematical notations like Z-notation and Vienna Development Method (VDM). CNLs maintain understandability, and mathematical notations offer precision in specifying data types and relationships. Visual representations such as Petri nets and Statecharts, along with logic-based methods like temporal logic, provide additional frameworks for requirement specification [70].

Tools like SPIN, NuSMV, and UPPAAL support formal verification, complemented by theorem provers such as Coq and Isabelle/HOL for specification correctness [71]. Specification languages like Alloy and Event-B provide frameworks for writing and analyzing formal specifications. TLA+ (Temporal Logic of Actions) exemplifies these formal approaches, offering precise specification of concurrent systems through temporal logic and set theory [72].

Consider a requirement stating "The engine RPM should remain above 2000 RPM." In TLA+, this requirement becomes:

```
VARIABLE rpm
Init == rpm >= 2000
Next == rpm' >= 2000
Spec == Init /\ [][Next]_<<rpm>>
```

'Init' defines the initial condition that the engine RPM must be at least 2000 RPM. 'Next' specifies that in every subsequent state, the RPM should remain above 2000 RPM, where rpm' represents the value of rpm in the next state. 'Spec' combines these conditions to ensure the requirement holds throughout the system's operation.

One popular formalism for specifying and reasoning about real-time systems is Metric Temporal Logic (MTL), which is an extension of Linear Temporal Logic (LTL) to add quantitative time constraints to temporal operators. The obvious constraint of LTL was in its inability to specify timing constraints, it is only able to express qualitative timing relationships like "Always" or "Eventually", MTL solves this by allowing system designers to express properties of the kind "event A must occur within 5 time units of event B" [73].

The basic syntax of MTL includes:

$$\Diamond_{[a,b]}\phi \quad \text{(Eventually: } \phi \text{ must become true within } [a,b])$$

$$\Box_{[a,b]}\phi \quad \text{(Always: } \phi \text{ must remain true throughout } [a,b])$$

$$\phi\, U_{[a,b]}\, \psi \quad \text{(Until: } \phi \text{ must remain true until } \psi \text{ becomes true within } [a,b])$$

Where ϕ and ψ are MTL formulas, and [a,b] represents a time interval. MTL formulas are typically interpreted over timed state sequences, which are sequences of states annotated with timestamps. The semantics of MTL define when a timed state sequence satisfies an MTL formula. For example, consider this requirement "Whenever the engine speed exceeds 5000 RPM, it must return to below 5000 RPM within 2 seconds." The MTL requirement can be written as:

$$\Box(\text{engine\_speed} > 5000 \rightarrow \Diamond_{[0,2]}(\text{engine\_speed} \leq 5000))$$

Because of MTL's ability to quantify over time in this manner, it has been applied in many fields and areas, including:

1. Specification of real-time systems: MTL is particularly useful in domains such as automotive systems, aerospace, and industrial control systems where precise timing is needed.

2. Formal verification: MTL formulas can be used as input to model checking tools to verify that a system design meets its timing requirements.

3. Runtime verification: MTL specifications can be used to generate verifiers that check system behavior during execution.

2.5.2 Requirement Verification and Representation

We employ formal verification to mathematically prove or disprove whether system designs meet their specifications [74]. Unlike traditional bug-finding methods, formal verification guarantees the absence of errors within verified properties. Bernardeschi et al. [75] demonstrated this in motor control systems using PVS (Prototype Verification System) to verify stability requirements. Grimm et al. [76] evaluated several formal verification techniques: Automated Theorem Proving (ATP), Symbolic Model Checking (SMC), Bounded Model Checking (BMC), Satisfiability Modulo Theory (SMT), Equivalence Checking, and Static Analysis. Each technique offers distinct advantages, from ATP's mathematical rigor to SMC's exhaustive state verification, though implementation requires significant expertise.

Simulation-based verification validates system behavior in complex designs, complementing formal verification techniques [77]. Formal methods provide exhaustive proofs, and simulations offer practical insights into system performance across diverse operating conditions. The effectiveness of simulation depends on careful parameter selection, particularly time step sizes. In powertrain simulations, engine combustion events may require microsecond-scale steps, but vehicle dynamics can be captured with millisecond-scale steps. Adaptive time-stepping algorithms, like Runge-Kutta-Fehlberg methods, optimize this trade-off between computational resources and accuracy. Operating conditions are explored through statistical sampling techniques such as Latin Hypercube Sampling or Sobol sequences. For hybrid electric vehicle powertrains, these conditions might include battery state of charge (20% to 80%), ambient temperature (-20°C to 40°C), and driver demand profiles. This simulation approach aligns naturally with digital twin concepts through SSP and FMI standards, where powertrain components are modeled as FMUs exchanging data at defined communication points. This modular structure enables continuous model refinement based on real-world data, supporting effective requirement verification and system optimization.

Ontologies complement these verification methods by providing a structured framework for requirement representation [78, 79]. Linear Temporal Logic (LTL) requirements can be represented as instances within a Requirement Ontology, maintaining formal semantics while enabling standardized expression. Tools like Protégé support creation and validation of these ontological structures [80], and the Semantic Web Rule Language (SWRL) provides additional rule definition capabilities. For example, a requirement that an event "p" should never occur (expressed in LTL as $G\neg p$) can be structured in an ontology, enabling clear communication between stakeholders and integration with verification tools through query languages like SPARQL [63].

2.6 Summary of Background

This background section established the foundations for addressing dynamic requirement verification in complex powertrain systems. Systems engineering has evolved into Model-Based Systems Engineering (MBSE) to handle increasing system complexity through model-based approaches. Digital twins represent the latest advancement in MBSE, enabling virtual system testing before physical implementation. Standards like SSP and FMI have enabled collaborative design through co-simulation and model exchange.

The fundamental challenge of formalizing dynamic, time-dependent requirements persists in these modern engineering approaches. Ontology-based approaches offer structured knowledge representation. Formal methods provide mathematical precision through languages like TLA+ and MTL. Current verification methods, from automated theorem proving to simulation-based testing, lack a comprehensive framework for handling time-bounded requirements in complex systems. This gap between requirement formalization and verification capabilities motivates this research. The following section presents a systematic review focusing on research efforts that combine formal logic with requirement specification in complex systems, particularly examining approaches that integrate ontologies with verification methods.

3 Literature Review

This chapter reviews recent research in formalizing dynamic requirements for complex engineering systems. Section 3.2 examines how ontologies structure and represent engineering knowledge within MBSE frameworks. Methods for verifying these formalized requirements through simulation and formal techniques are explored in Section 3.3. Integration of verification approaches into virtual system representations through digital twins are investigated in Section 3.4. Broader technological integration efforts are examined in Section 3.5. The findings are synthesized in Section 3.6 to identify current limitations and opportunities in powertrain system verification.

3.1 Review Methodology and Results

Dynamic requirement verification in simulation-driven collaborative design of powertrains continues to attract research attention. This section presents a systematic review methodology to capture recent developments and identify current research trends, complementing the background information from preceding sections. The objectives of this review are:

1. To examine recent advancements in ontology-based approaches for representing dynamic requirements in systems engineering, particularly within powertrain design.

2. To evaluate contemporary simulation-driven verification methods for dynamic requirements in collaborative design environments.

3. To assess the integration of SSP and FMI standards in co-simulation frameworks for complex powertrain systems.

4. To identify research gaps and propose future directions in dynamic requirement verification, emphasizing ontology-based representations.

This systematic review follows the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) guidelines [81]. The review process consists of three main stages: literature identification, filtering based on defined criteria, and information extraction and interpretation. The findings inform subsequent chapters and contextualize this research within broader scientific discourse.

This review searched IEEE Xplore, Scopus, and Web of Science databases in April 2024, chosen for their comprehensive coverage of engineering and computer science literature. The search string combined several concepts:

```
(ontolog*) AND
("dynamic requirement*" OR "requirement* verification" OR verification) AND
(simulation OR "digital twin*") AND
(design OR "system* engineering") AND
(powertrain* OR "maritime system*" OR "ship system*" OR "marine engineering"
OR engineering)
```

Based on this search strategy, the review examined peer-reviewed articles published in English between 2000 and 2024, selecting papers that met specific inclusion and exclusion criteria. Papers were included if they addressed:

- Ontology-based approaches in MBSE requirement formalization.
- Simulation and verification methods for complex engineering systems.
- Dynamic requirement verification in design processes.
- Collaborative design environments.

Papers were excluded if they:

- Lacked connection to engineering design or systems engineering.
- Had no full text available.

Selected papers were first grouped thematically based on their core contributions. The analysis examined advances and limitations in three areas: requirement formalization through ontologies, simulation methods, and collaborative design practices. Special attention was given to approaches applicable to powertrain systems.

The systematic review process identified 112 initial papers, with 28 papers selected for final analysis following the PRISMA methodology (Figure 12). These peer-reviewed articles span multiple research areas including ontology development, simulation techniques, and digital twin implementation. While the search covered publications from 2000 to 2024, papers published before 2015 were incorporated into the background chapter to provide foundational concepts. The systematic review focuses on post-2015 papers, representing diverse domains including powertrain systems, maritime systems, automated driving systems, and aircraft manufacturing.

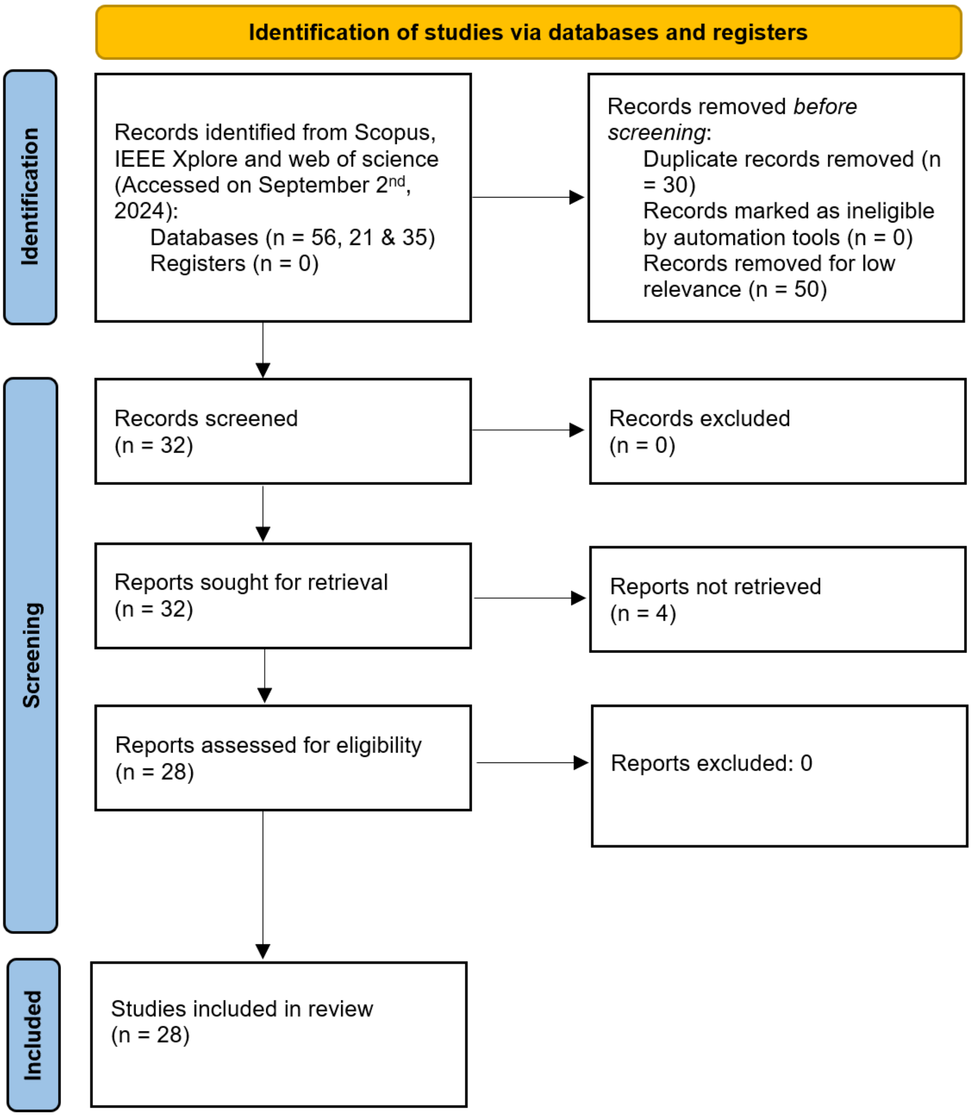

Figure 12: PRISMA flow diagram of the study selection process.

The analysis reveals recent research approaches to formal requirement representation in systems engineering. Section 3.2 examines foundational work in using ontologies to structure engineering requirements. This formalization enables the verification methods explored in Section 3.3, where simulation and formal techniques validate system behavior. Section 3.4 shows how these approaches manifest in digital twin implementations, while Section 3.5 examines efforts to unify these methods in practical engineering workflows. Section 3.6 synthesizes these developments to identify opportunities in dynamic requirement formalization and verification.

3.2 Ontology-based Approaches in MBSE

Ontology-based approaches are used in Model-Based Systems Engineering (MBSE) due to their ability to formalize domain knowledge and improve model consistency. Several contributions have been made, each building upon previous work and addressing different aspects of ontological modeling in systems engineering.

Guizzardi et al. [82] laid a foundational method with the Unified Foundational Ontology (UFO), a framework for ontology-driven conceptual modeling. UFO comprises three main parts: UFO-A (endurants), UFO-B (perdurants), and UFO-C (social and intentional concepts). The authors demonstrated UFO's application in analyzing, re-engineering, and integrating modeling languages across different domains. They also introduced OntoUML, an ontologically well-founded version of UML class diagrams based on UFO-A. The paper highlights the iterative development of UFO and OntoUML based on practical feedback, bridging theory and practice in ontology-driven conceptual modeling. UFO has become a well-established foundational ontology specifically designed for conceptual modeling.

Guerson and Almeida [83] expanded OntoUML's capabilities to better represent dynamic aspects of systems. They developed a temporal extension of OCL (Object Constraint Language) based on a concept called "world state reification." This approach allows modelers to express complex time-based constraints in their system models. It supports dynamic classification, meaning it can represent how objects change their types over time, and modal constraints, which describe what's possible or necessary in a system. This addresses a need in system modeling, especially for systems that change over time. However, this approach has some limitations. The world-state model they use can't directly represent specific time intervals, which is a problem when modeling real-time systems that need precise timing. Moreover, using OCL for complex time relationships can lead to long, complicated expressions that are hard to read and understand. The approach also lacks a strong mathematical foundation that more rigorous time-based logics have. This makes it harder to prove that the model is correct. Lastly, the authors do not address how well this method works for large, complex systems with many interconnected time-based constraints, which could be a practical issue in real-world applications.

Guizzardi et al. [82] and Guerson and Almeida [83] focused on broad conceptual modeling. In contrast, Kabaale et al. [84] applied ontological principles specifically to software development processes. They created a new model based on four main ideas: Achievable, Doable, Tangible, and Assessable. These ideas aim to help automate, adapt, and verify software development processes more effectively. Their model uses a technique called the powertype pattern, which allows for flexible customization of processes. They formalized this model using Web Ontology Language (OWL), a language designed for creating and working with ontologies. Their approach uses a concept called "clabjects," which combines two ways of organizing information: grouping similar things (instantiation) and creating more specific categories (generalization). This axiom-based approach, combined with ontology formalization, makes software processes easier to verify. It allows for checking using special reasoning software designed for ontologies. However, the authors note that more testing in real-world applications is needed to prove the approach's utility in practice. Their use of OWL to formalize abstract concepts provides insights for requirement representation, particularly their method of bridging different abstraction levels through clabjects - a relevant consideration when formalizing requirements that must connect system-level specifications to component-level behaviors.

Sirin et al. [85] developed the Model Identity Card (MIC) approach for standardizing simulation model specifications in cross-discipline environments. Their approach establishes a common vocabulary between Original Equipment Manufacturers (OEMs) and model providers through structured model attributes and interface definitions. While they criticize SysML's steep learning curve, their solution relies on arKItect - another specialized tool requiring significant training. Their introduction of a "Model Architect" role to manage model interfaces and connections adds organizational complexity that may be impractical for many teams. Their method successfully demonstrates how structured vocabularies can reduce ambiguity in model specifications, particularly in interface definitions. However, their validation using only automotive case studies limits broader applicability, and their interface-focused approach overlooks temporal aspects of model behavior. Their structured attribute classification provides insights for requirement formalization, though their emphasis on tool-specific implementation rather than underlying formalization principles limits its theoretical foundation.

Chebieb and Ait-Ameur [86] formalized user interface plasticity verification by combining labeled state-transition systems (LTS) with domain ontologies. Their approach uses bi-simulation relationships to verify equivalence between different interaction modes, with ontologies providing the semantic foundation for comparing states. Their methodology demonstrates how formal verification can be accomplished through ontological reasoning - particularly in establishing equivalence between different interface implementations. Their work focuses on stateless interface verification, however, it provides methods for formalizing dynamic requirements. Their use of ontologies to establish relationships between different interaction modes offers a pattern for connecting abstract requirements to concrete implementations. Nevertheless, their bi-simulation approach lacks support for temporal properties and their ontology requires significant domain expertise to develop and maintain.

These ontology-based approaches demonstrate how domain knowledge can be rigorously formalized through mathematical logic. The progression from purely conceptual modeling to specific domain applications like user interface verification shows ontologies' potential for precise knowledge representation. Particularly, works combining ontologies with formal methods establish a foundation for verifiable system properties. However, these approaches also reveal challenges - most notably in handling temporal specifications and dynamic behaviors. This gap between static ontological representations and dynamic system requirements motivates the need for specialized temporal extensions in requirement formalization.

3.3 Simulation and Verification Methods in Systems Engineering

Research into advanced simulation and verification methods aims to streamline the design-to-verification process. These methods enable more accurate models, better iterative designs, and faster project development. Savolainen et al. [87] developed a framework for runtime verification of industrial process control systems. Their approach combines keyword-driven testing with ontologies to enable framework compatibility across different simulators. Through a case study in mineral processing, their service-oriented architecture identified design issues in both process and automation domains, particularly in control loops and instrumentation. The framework's effectiveness depends on simulation model accuracy, presenting opportunities for integration with formal verification methods. Though the authors note limited support for manufacturing industry tools, their ontology-based approach demonstrates how semantic technologies can standardize verification across different simulation environments. The runtime verification focus suggests potential for combining simulation-based testing with formal requirement specification to achieve more comprehensive verification coverage.

Building on the concept of service-oriented architectures, Lu et al. [88] developed a tool-chain to support MBSE of aero-engines. Their approach combines domain-specific modeling with open standards like Open Services for Lifecycle Collaboration (OSLC) and Functional Mock-up Interface (FMI) to promote interoperability. The tool-chain transforms domain-specific models into ontologies to generate OSLC services and a web-based process management system. Through a case study of aero-engine performance analysis, they demonstrated how their approach reduced manual operations by 85% compared to traditional methods. Their implementation relies on custom XML schemas generated from domain-specific models rather than established ontology standards like OWL, limiting semantic interoperability and ontology reuse. Their work demonstrates the potential of combining service-oriented architectures with ontologies for simulation automation, though their approach to modifying both meta-models and ontology schemas for extensions contradicts established ontology design principles.

Previous work focused on implementation architectures, subsequent research explored quality assessment frameworks. Bossa et al. [89] proposed a methodology for co-simulation based model assessment in MBSE for complex systems in Extended Enterprise environments. Their approach introduces Simulation Quality Requirements (SQRs) derived from System Architecture Requirements, focusing on three model qualities: accuracy (closeness to theoretical values), representativeness (abstraction level of physical phenomena), and stability (robustness to input perturbations). The work remains largely theoretical, however, their quality framework provides valuable insights for formalizing simulation requirements. They identify the need for domain-specific ontologies to express physical phenomena more precisely than previous approaches, though they do not fully develop these ontologies. The authors propose flexible strategies for grouping system functions into FMUs, acknowledging the balance between intellectual property protection and system integration needs.

Kibret et al. [90] applied category theory, a branch of mathematics, to formalize the Verifiable Design Process (VDP) in systems verification. Their approach represents VDP components as categories and uses these structures to analyze relationships between different system representations. The VDP is made up of stacked abstraction layers. Each layer has requirements in natural language (L_n), requirements in ontological form (L_o), specifications in logic form with associated proofs (M_l and T), and systems of coupled models and simulations ($M_{S/P}$ and $S_{S/P}$). The authors define four main categories: requirements (L_n), ontology (L_o), logical representation (L_l), and systems of interconnected models (M_s). They use mathematical concepts like pullbacks and pushouts to analyze relationships between these representations. A contribution is representing each abstraction layer as a category of categories, with individual representation of categories as objects and mappings between them with arrows. This allows for formal analysis of the entire design process and each abstraction level. This work provides a strong mathematical foundation for analyzing system representations and formalizes refinement and abstraction processes. However, the work remains largely theoretical. The authors do not provide details on practical implementation, particularly regarding how the mappings between different representations can be completed in a reliable way.

Elgharbawy et al. [91] developed an adaptive verification framework for automated driving systems that combines ontology-based test scenario synthesis with data mining. Their framework uses ontologies to transform natural language requirements into formal specifications through a layered approach: structured natural language, ontological representations, and logical specifications. They demonstrate this through a collision-free emergency braking system case study, where they successfully identify and classify different driving scenarios using hierarchical clustering. Their approach to ontology development separates class-level knowledge (general concepts and rules) from instance-level data (specific scenarios and observations), following standard ontology practices. The framework includes practical considerations for implementation, such as criticality metrics and scenario synthesis. Though their approach shows promise for formalizing dynamic requirements, they acknowledge limitations in automating variable selection and require manual interpretation of clustering results. The framework's reliance on Hardware-in-the-Loop simulation also raises questions about scalability for more complex scenarios.

Cibrián et al. [92] developed PhysicalModel2Simulink, a solution for reusing knowledge in Simulink models through semantic representation. Their approach introduces a Physical Model Mappable Element (PMME) that standardizes how physical components are represented, independent of the modeling language. The PMME captures both meta data and model contents using a common domain vocabulary and taxonomy, creating a property graph for system artifact discovery. Their case study with 38 Simulink models achieves high precision (77%) and recall (89%) in model retrieval. Their paper provides limited details about ontology structure and design decisions. A significant limitation is the lack of standardization in search terms - engineers have no guidance on terminology, an issue that could have been addressed through domain-specific ontologies. The validation uses only a small set of Simulink models, and extending the approach to other physical modeling tools would require significant rework. Their results demonstrate the potential of ontology-based approaches for model retrieval, but practical application would require a more rigorous ontological foundation.

Recent work has increasingly focused on practical implementation challenges. Chen et al. [93] presented a service-oriented approach to model integration in MBSE, focusing on the integration between architecture design and system verification. Their method uses Open Services for Lifecycle Collaboration (OSLC) to transform heterogeneous models and APIs into unified web-based services with defined semantic mapping rules. Their mapping approach successfully handles semantic differences between models, but requires manual development of adapters for each model type. The authors acknowledge some limitations: models cannot be created through OSLC services, and significant effort is needed to develop adapters for different tools. Their work demonstrates both the potential and challenges of using service-oriented architectures for model integration - while semantic mappings can bridge different representations, the implementation remains complex.

These studies demonstrate the evolution of simulation and verification methods in MBSE from 2017 to 2023. A clear trend emerges towards integrated approaches that combine formal methods, ontologies, and service-oriented architectures to address model heterogeneity and verification challenges. Semantic technologies and standardized interfaces have improved interoperability. Yet some challenges persist: scalability of verification methods, complexity of practical implementation, and the need for comprehensive validation in real-world scenarios [94]. These limitations inform the approach taken in this thesis for formalizing dynamic requirements.

3.4 Digital Twins and Advanced Modeling Techniques

Digital twins support collaborative design by providing virtual environments where multiple teams can integrate and test components [95]. In complex systems development, digital twins enable concurrent simulation of components from different manufacturers, supporting both design verification and system integration. Kovalyov [96] proposes a theoretical framework for power system digital twins using ontologies. The framework outlines the integration of ontology models, graphical representations, digital records, data structures, live information streams, and simulation models through mathematical concepts. The work remains at a conceptual level without practical implementation, yet demonstrates increasing interest in ontology-based approaches for complex system integration. The framework suggests methods for standardizing information exchange between components, including requirements and specifications.

Building on the integration potential of digital twins, Madni [97] developed an MBSE testbed to address growing complexity in system lifecycle management. The testbed integrates digital twins with verification tools through a cloud-based framework, combining scenario builders, model repositories, and simulation engines. Their implementation demonstrates practical applications in aircraft security systems and autonomous vehicles, showing how digital twins support system verification. The testbed enables comparing different control architectures and evaluating system behavior through simulation-driven dashboards.

Lu et al. [98] and Zheng et al. [99, 100] demonstrated how ontologies can transform formal specifications into verifiable simulation models. Their work in semantic Model-based Systems Engineering (sMBSE) applies this approach to manufacturing processes. Using KARMA (Kombination of ARchitecture Model specificAtion), they capture system specifications across five perspectives: mission, operation, function, logic, and physical structure. The ontological framework transforms these specifications into executable models - Petri Nets in Lu's work and Discrete Event Simulations in Zheng's approach. This transformation streamlines verification and improves knowledge integration across design stages. Zheng et al. [100] further extend this concept through a semantically-guided design space for aircraft systems, where their ontology merges assembly expertise, industry requirements, and structural specifications. Their success in manufacturing demonstrates how ontology-based approaches can bridge the gap between system specification and verification.

These ontology-based approaches reflect a growing trend in systems engineering toward integrated design and simulation environments. The transformation of formal specifications into verifiable models, demonstrated in manufacturing domains, shows how ontologies can standardize information exchange between different stakeholders. This standardization becomes essential in complex systems involving multiple development teams. The approaches succeed in integrating domain knowledge with simulation models. The addition of temporal logic or other formal methods to these ontological frameworks could provide more rigorous verification of dynamic system behavior.

3.5 Ontology-Based Integration in MBSE

Complex systems development requires integration between different modeling tools and frameworks. MBSE approaches increasingly incorporate technologies that support collaborative design. Lu et al. [101] address this need through an ontology-based framework for co-simulation in MBSE. Their scenario-based ontology formalizes integration between Domain-Specific Modeling (DSM) models, technical resources, and process management systems. Using Web Ontology Language (OWL), they develop an ontology that connects Open Services for Lifecycle Collaboration (OSLC) services with Service Orchestration Templates (SoTs).

Their framework supports model exchange through Functional Mock-up Interface (FMI) standards, enabling integration between different simulation tools. Through SPARQL and SQWRL queries in a MATLAB/Simulink case study, they demonstrate how ontologies can standardize information exchange between modeling tools and support co-simulation processes.

The evolution of MBSE shows increasing focus on integrating collaborative design methods with formal specification approaches. Frameworks combining ontologies, co-simulation standards, and system modeling tools provide structured methods for knowledge representation and verification. These integration efforts reflect the growing need to bridge gaps between different domains and stakeholders. The combination of ontological frameworks with formal specifications points toward solutions for dynamic requirement representation. This approach becomes particularly relevant for complex systems where time-dependent behaviors need precise verification through simulation.

3.6 Research Gaps

The past decade has advanced collaborative design through ontology-based approaches, simulation methods, and digital twins. In simulation-driven collaborative design of powertrains, several research gaps emerge that shape the research questions stated in Section 1.2:

1. Current approaches lack methods for representing and verifying dynamic requirements with temporal constraints in collaborative environments. Formal frameworks must capture time-dependent behaviors involving multiple stakeholders. This necessitates research into uniform integration methods for dynamic requirements and their verification through simulation.

2. Ontologies support knowledge representation in collaborative design but remain disconnected from formal methods for requirement specification. This separation limits the ability to verify requirements across different domains and stakeholders. Research must examine how ontology-based frameworks can standardize system descriptions across different sources.

3. Verification frameworks handle mainly static requirements or simplified dynamic behavior. Complex systems like powertrains need requirement verification that accounts for interactions between components from different providers. Understanding how simulation-based verification can verify these requirements becomes critical.

4. Design tools struggle to maintain requirement consistency during system development, especially when components come from multiple sources. Methods for managing evolving requirements in collaborative environments remain underdeveloped. This raises questions about mapping component-defined variables to standardized requirement interpretation systems.

5. Simulation-driven design lacks clear connections between component specifications and verification activities. The gap between component-level requirements and system-level verification hinders collaborative development. Research must address both system architecture standardization and the mapping between component specifications and requirements.

This thesis primarily addresses research gaps (1), (2), and (3) through an ontology-based framework that combines Metric Temporal Logic with semantic web technologies. The framework enables verification of dynamic requirements in collaborative powertrain design, focusing on requirements that can be formalized in MTL and verified through simulation. Gap (4), concerning requirement evolution throughout system lifecycle, and aspects of gap (5) related to continuous integration of verification activities remain as future work. Chapter 4 presents the methodology that bridges design and verification processes through this framework, demonstrating its application in powertrain systems while acknowledging these scope limitations.

4 Methodology

This research develops an ontology to represent Metric Temporal Logic (MTL) for requirement verification, building on verification methods established in Siivola's thesis [102]. The requirements are tested in powertrain systems, leveraging Semantic Web technologies to bridge system-level specifications and component-level requirements. The approach integrates System Structure and Parameterization (SSP) and Functional Mock-up Interface (FMI) concepts.

4.1 Ontology Overview

The ontology is developed using the Web Ontology Language (OWL) and is serialized in Turtle (.ttl) format. It incorporates concepts from several existing ontologies and standards. As described in 2.4, ontologies can be reused and integrated. This property was leveraged in this thesis; application ontologies like Vehicle Signal Specification Ontology VSSo, generic ontologies like SOSA/SSN and OWL-Time were used to capture the temporal concepts defined in MTL.

- Vehicle Signal Specification Ontology (VSSo) for automotive-specific concepts, many of which extend to marine powertrain concepts.
- Semantic Sensor Network (SSN) and Sensor, Observation, Sample, and Actuator (SOSA) ontologies for generic representation relationships not defined in VSSo.
- Time Ontology for temporal concepts in MTL.
- QUDT ontology for units of measurement.

Metric Temporal Logic (MTL) extends Linear Temporal Logic (LTL) to specify properties constrained by time, enabling precise requirement definitions in dynamic systems. The concepts of temporal logic and MTL presented here are based on the work of Bellini et al. [103]. Building on the introduction in Section 2.5.1, we first establish the formal mathematical foundations of MTL to provide necessary context for understanding how temporal logic concepts are captured in the ontology.

In temporal logic, we need a way to represent events occurring over time. We achieve this using the concept of "timed words", which combine events with their timestamps. An alphabet *alphabet* (Σ) defines the set of events or states in the system. In a powertrain system, this alphabet might contain events like "engine start", "gear shift", or states such as "engine speed > 1000 RPM". A *word* represents a sequence of these events or states as they occur, while a *timestamp* indicates precisely when each event occurs or when each state is observed.

Let $\mathbb{R}_{\geq 0}$ denote the set of non-negative real numbers. A time sequence must satisfy three properties: initialization (starting at time zero, $\tau_0 = 0$), monotonicity (each time value is greater than or equal to the previous one, $\tau_i \leq \tau_{i+1}$ for $i = 0, 1, ..., n - 1$ where n is the sequence length), and progress (for infinite sequences, time grows without bound, $\{\tau_i : i \in \mathbb{N}\}$ is unbounded).

A timed word combines a sequence of events with their corresponding timestamps:

$$\rho = (\sigma, \tau), \text{ where } \sigma = \sigma_0\sigma_1 \dots \text{ and } \tau = \tau_0\tau_1 \dots \tag{1}$$

To illustrate this concept in the context of a powertrain system:

$$\rho = ((engineStart, 0.0), (speed > 1000, 2.5), (gearShift, 3.7), \dots) \tag{2}$$

This represents the engine starting at time 0.0, reaching 1000 RPM at time 2.5, and shifting gears at time 3.7.

Generally, for an alphabet $\Sigma = \{a, b\}$, a timed word takes the form:

$$\rho = ((a, 0.5), (b, 1.2), (a, 3.7), \dots) \tag{3}$$

where each pair represents an event and its timestamp. This structure allows us to precisely track when events occur or when specific states are observed, as illustrated in Figure 13.

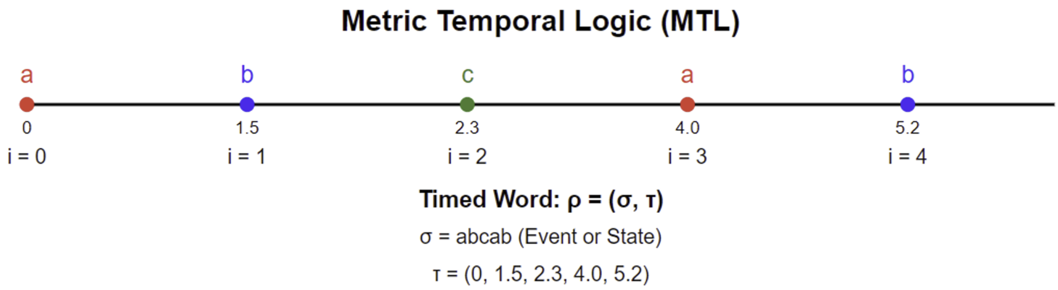

Figure 13: Illustration of the structure of MTL, a timed word representing events at each time step, and the index between each timed word.

Given a set of atomic propositions AP, the syntax of MTL formulas is defined inductively as:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathbf{U}_I \varphi_2 \tag{4}$$

where $p \in AP$, $\varphi, \varphi_1, \varphi_2$ are MTL formulas, and $I \subseteq \mathbb{R}_{\geq 0}$ is an interval with endpoints in $\mathbb{N} \cup \{\infty\}$. For a timed word $\rho = (\sigma, \tau)$ at position $i < |\rho|$, the satisfaction relation $(\rho, i) \models \varphi$ defines when an MTL formula φ holds:

$$\begin{aligned}
(\rho, i) &\models \top && \text{always} \\
(\rho, i) &\models p && \text{iff } p \in \sigma_i \\
(\rho, i) &\models \neg\varphi && \text{iff } (\rho, i) \not\models \varphi \\
(\rho, i) &\models \varphi_1 \wedge \varphi_2 && \text{iff } (\rho, i) \models \varphi_1 \text{ and } (\rho, i) \models \varphi_2
\end{aligned} \tag{5}$$

The Until operator defines temporal relationships between formulas:

$$\begin{aligned}
(\rho, i) \models \varphi_1 \mathbf{U}_I \varphi_2 \quad &\text{iff there exists } j > i \text{ such that } (\rho, j) \models \varphi_2, \\
&\tau_j - \tau_i \in I, \text{ and} \\
&(\rho, k) \models \varphi_1 \text{ for all } k \text{ with } i < k < j
\end{aligned} \tag{6}$$

An MTL formula φ is bounded if every interval I appearing in φ is bounded, meaning it can only express properties within a fixed, finite time horizon. Common temporal operators can be defined as abbreviations:

$$\begin{aligned} \text{Eventually: } & \Diamond_I \varphi \equiv \top \mathbf{U}_I \varphi \\ \text{Always: } & \Box_I \varphi \equiv \neg \Diamond_I \neg \varphi \\ \text{Next: } & \bigcirc_I \varphi \equiv \bot \mathbf{U}_I \varphi \end{aligned} \tag{7}$$

The ontology developed in this research allows for the representation of powertrain systems, their components, and associated requirements using Metric Temporal Logic (MTL). The ontological structure is built from several classes and their relationships to represent the domain.

This formalization of MTL provides the mathematical foundation for expressing real-time properties in our ontology. The verification approach used in this thesis builds upon the work of Siivola [102], where $\wedge$ and $\oplus$ are defined as set operators rather than binary operators, with conventions $\{\} = \text{true}$ and $\oplus\{\} = \text{true}$. The implementation uses only future-time temporal operators and employs term rewriting instead of finite state machines, making it suitable for iterative design processes.

The verification treats each (variable, time) pair as a timed word, enabling both real-time and post-simulation requirement checking. Building upon these concepts, this thesis develops an ontology that represents powertrain systems, their components, and requirements using MTL. The following sections detail the ontological structure and its implementation.

4.2 Requirement Ontology Module

The requirement ontology module is the core of the overall ontology structure, focusing on the representation and management of system requirements. This module was developed following established ontology design principles, including modular design, clear documentation, and reuse of existing ontologies where applicable.

The ontology uses Turtle (.ttl) files, a serialization format for Resource Description Framework (RDF) data [104, 105]. The World Wide Web Consortium developed Turtle to provide a compact and human-readable syntax for RDF data. This format enables integration of external ontologies and supports conversion to other structures such as eXtensible Markup Language (XML).

The development of the Requirement Ontology Module began by identifying terms and concepts related to requirements in powertrain systems. The core concepts include Requirement, Requirement Set, Conditional Requirement, and Default Requirement Set. These concepts form a single module that represents requirements at both system and component levels through the “req:” class hierarchy, as shown in Figure 14 below.

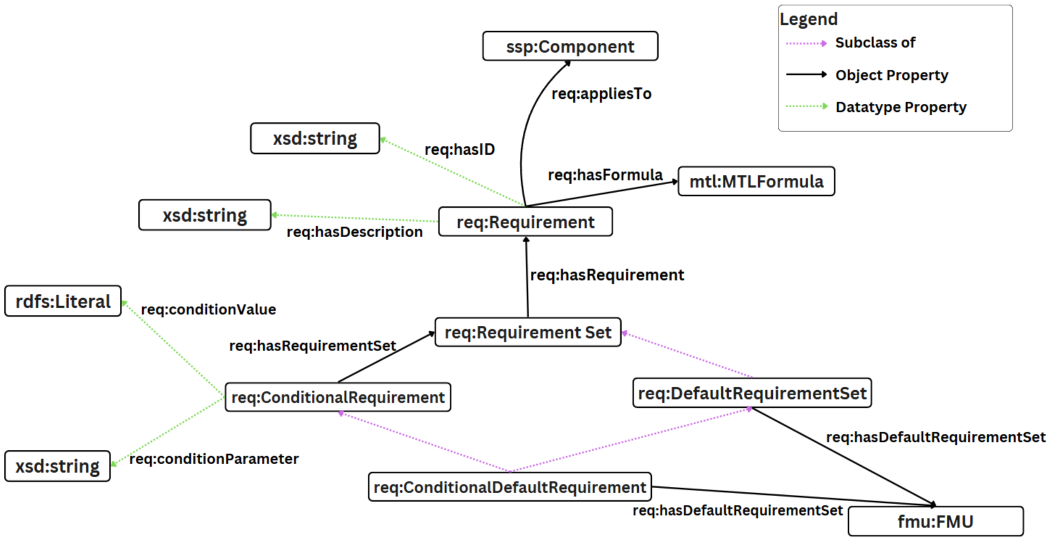

Figure 14: Requirement class hierarchy showing relationships and properties, detailing the requirement type, its ID, etc.

The `req:Requirement` class represents requirements at both system and component levels in powertrain systems. This class captures both formal and informal requirements through four properties: `req:hasFormula` for MTL representations, `req:hasDescription` for human-readable content, `req:hasID` for unique identifiers, and `req:appliesTo` for component relationships. This modular property structure follows common ontological design principles [106].

To handle powertrain specifications, the ontology extends the basic requirement class with additional structures. The `req:RequirementSet` class groups related requirements that pertain to specific operational modes or subsystems. For context-dependent behaviors, `req:ConditionalRequirement` uses `req:conditionParameter` and `req:conditionValue` to specify when requirements apply. Default behaviors are managed through `req:DefaultRequirementSet` and `req:ConditionalDefaultRequirement`, which define baseline requirements that can be overridden by more specific conditions.

The ontology connects these requirement structures to the broader system through two relationships. The `req:appliesTo` property links requirements to system components, while `req:hasDefaultRequirementSet` associates FMUs with their default requirements. These connections, implemented using OWL constructs (`owl:Class`, `owl:ObjectProperty`, `owl:DatatypeProperty`), enable representation of requirements ranging from simple constraints to complex, context-dependent specifications.

4.3 MTL Formula Module

The "mtl:" class represents Metric Temporal Logic formulas for expressing temporal properties of powertrain systems. Figure 15 shows the three main classes that define formulas, operators, and their combinations.

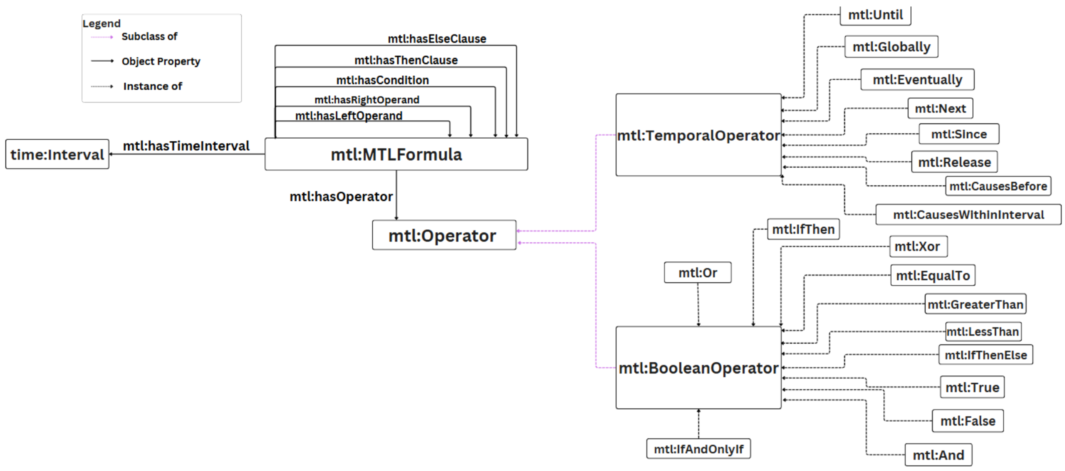

Figure 15: MTL class structure showing formula definitions, operators, and operator combinations.

The `mtl:MTLFormula` class enables formula expression through recursive composition. Its properties include `mtl:hasOperator` for linking to logical or temporal operators, `mtl:hasLeftOperand` and `mtl:hasRightOperand` for binary operations, and `mtl:hasTimeInterval` for specifying time bounds. Additional properties (`mtl:hasCondition`, `mtl:hasThenClause`, `mtl:hasElseClause`) support conditional expressions. This recursive structure mirrors the natural composition of MTL formulas, enabling straightforward transformation between ontological representations and executable MTL expressions through SPARQL queries.

Two operator subclasses form the core of MTL representation. The `mtl:TemporalOperator` includes time-bound operators like `mtl:Until`, `mtl:Globally`, and `mtl:Eventually`, while `mtl:BooleanOperator` provides logical operators (`mtl:And`, `mtl:Or`, `mtl:Not`) and comparisons (`mtl:EqualTo`, `mtl:GreaterThan`). Causal relationships are captured through `mtl:CausesBefore` and `mtl:CausesWithinInterval` operators. The `req:hasFormula` property connects these MTL formulas to system requirements, maintaining a clear separation between informal descriptions and formal specifications.

4.4 SSP and FMU Class

The ontology extends the System Structure and Parameterization (SSP) and Functional Mock-up Interface (FMI) standards to enable requirement representation. Figure 16 shows how SSP and FMU classes are structured to link systems and components with their requirements.

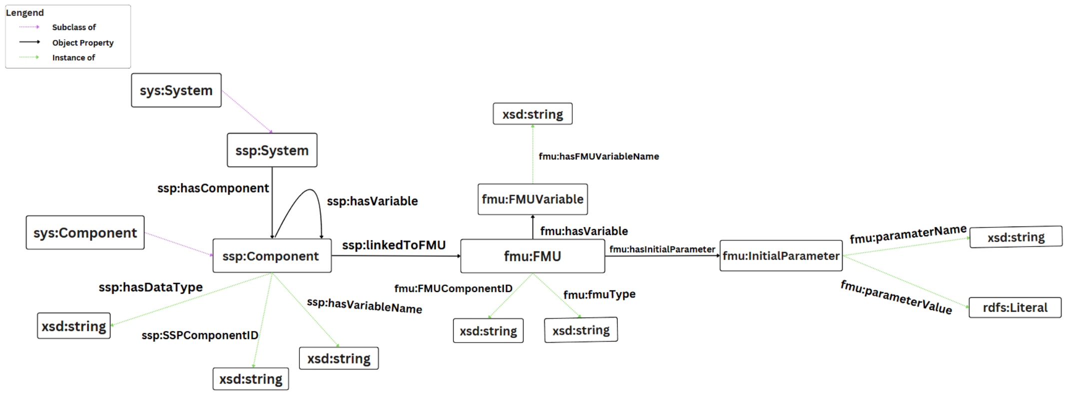

Figure 16: SSP and FMU class structure showing system definition and component relationships.

The `ssp:System` class represents the overall system structure, connecting to components through `ssp:hasComponent`. Each component has identifying properties (`ssp:componentType`, `ssp:SSPComponentID`) and variable definitions (`ssp:hasVariableName`, `ssp:hasDataType`).

The `fmu:FMU` class represents individual Functional Mock-up Units, each identified by `fmu:FMUComponentID` and type. Component variables are managed through the `fmu:FMUVariable` class, while `fmu:initialParameter` enables conditional requirement selection based on an initialized parameter. For instance, an engine's requirements might vary based on its size parameter.

The `ssp:linkedToFMU` property connects SSP components to their corresponding FMUs, maintaining consistency between system-level descriptions and component implementations.

4.5 Requirement Definitions Using the Ontology

This section demonstrates how requirements are formalized using the defined ontology, integrating concepts from System Structure and Parameterization (SSP) and Functional Mock-up Interface (FMI) standards. The ontological representation bridges human-readable specifications and machine-processable formats for verification within the Simantics platform.

The ontology employs a recursive structure to define Metric Temporal Logic (MTL) formulas, enabling representation of complex logical and temporal relationships. Each MTL formula can contain sub-formulas as operands through nested definitions. This structure maintains links between requirements, system components, and their variables, connecting logical requirements to the physical components they govern.

Another aspect of this approach is the alignment of variable names across different domains. The variables referenced in both system and component-level requirements must correspond exactly to those used in the simulation models, as defined in the Functional Mock-up Units (FMUs) and the SSP structure. It's important to note that within the SSP structure, variables must be referred to by their complete path, reflecting their position in the system hierarchy.

System-level requirements for ship powertrains define the overall performance and behavior expectations for the entire system. Our ontology-based approach enables structured representation of these requirements, linking natural language descriptions with formal Metric Temporal Logic (MTL) formulas. Figure 17 shows how a ship powertrain system is defined in our ontology, with six main components each representing a singular FMU model.

```
# Ship Powertrain System
sys:ShipPowertrainSystem a ssp:System ;
    rdfs:label "Ship Powertrain System" ;
    rdfs:comment "Represents the complete ship powertrain system" ;
    ssp:hasComponent sys:SSPController, sys:SSPHull, sys:SSPGearbox, sys:SSPDieselEngine, sys:SSPPropeller,
    sys:SSPConsumptionBlock .
```

Figure 17: Ship powertrain system definition showing the main system structure and its six component FMUs. System-level requirements can target the whole system, individual components, or specific variables.

Figure 18 demonstrates component and variable definition using the hull as an example. The hull integrates SSP component definitions with SOSA observable properties, while its attributes leverage Vehicle Signal Specification Ontology (VSSo) concepts like `vsso:Speed` and `vsso:Acceleration`. This approach combines domain-specific concepts with general IoT sensor standards.

```
# ship hull
sys:SSPHull a ssp:Component, sosa:ObservableProperty ;
    rdfs:label "SSP Hull" ;
    ssp:componentType "Hull" ;
    ssp:SSPComponentID "HULL_001" ;
    ssp:hasVariable sys:SSPHullSpeed, sys:SSPHullAcceleration  .

sys:SSPHullSpeed a vsso:Speed ;
    rdfs:label "Hull_Speed" ;
    ssp:hasVariableName "SSPHullSpeed_001" ;
    qudt:unit unit:Knot ;
    ssp:hasDataType xsd:double .

sys:SSPHullAcceleration a vsso:Acceleration ;
    rdfs:label "Hull_Acceleration" ;
    ssp:hasVariableName "SSPHullAcceleration_001" ;
    qudt:unit unit:Knot ;
    ssp:hasDataType xsd:double .
```

Figure 18: Component and variable definitions for the hull, showing integration of SSP components with SOSA properties and VSSo concepts. Each variable includes name, unit, and data type specifications, with extension capabilities through SOSA/SSN relationships.

When VSSo lacks specific component definitions, the ontology leverages SOSA/SSN capabilities, using `ssn:isPropertyOf` relationships to define new properties (examples in appendix B). This thesis uses VSSo and SOSA due to their sufficient coverage of powertrain terminology, though the ontology structure supports integration with additional domain-specific or top-level ontologies as needed.

With the system and components defined, we can now represent system-level requirements. Figure 19 shows a requirement definition with its name, unique ID, and associated components. Unique component IDs enable precise variable location in complex systems, such as ships with multiple engines.

```
req:ShipAccelerationRequirement a req:Requirement ;
    req:hasID "SHIP_REQ_002" ;
    req:hasDescription "The ship powertrain system shall accelerate from 0 to 15 knots within 5 minutes under calm water
    conditions." ;
    req:hasFormula req:ShipAccelerationFormula ;
    req:appliesTo sys:SSPHull .
```

Figure 19: Ship acceleration requirement definition showing requirement properties and component relationships. Unique IDs ensure unambiguous reference to components and their variables in the SSP model.

Natural language requirements must be formalized for verification. Figure 20 shows how Metric Temporal Logic expresses this requirement as a mathematical formula, combining temporal operators with system variables to enable machine verification.

```
req:ShipAccelerationFormula a mtl:MTLFormula ;
    mtl:hasOperator mtl:CausesWithinInterval ;
    mtl:hasTimeInterval [
        a time:Interval ;
        time:hasBeginning [
            a time:Instant ;
            time:inTimePosition [ time:numericPosition "0"^^xsd:double ; time:hasTRS time:unitSecond ]
        ] ;
        time:hasEnd [
            a time:Instant ;
            time:inTimePosition [ time:numericPosition "300"^^xsd:double ; time:hasTRS time:unitSecond ]
        ]
    ] ;
    mtl:hasLeftOperand [
        mtl:hasOperator mtl:True ;
        mtl:hasRightOperand [
            a mtl:MTLFormula ;
            mtl:hasOperator mtl:GreaterThan ;
            mtl:hasLeftOperand sys:SSPHullSpeed ;
            mtl:hasRightOperand [
                a mtl:MTLFormula ;
                mtl:hasOperator mtl:EqualTo ;
                mtl:hasRightOperand "0"^^xsd:double ;
                qudt:unit unit:knots
            ]
        ]
    ] ;
    mtl:hasRightOperand [
        mtl:hasOperator mtl:True ;
        mtl:hasRightOperand [
            a mtl:MTLFormula ;
            mtl:hasOperator mtl:LessThan ;
            mtl:hasLeftOperand sys:SSPHullSpeed ;
            mtl:hasRightOperand [
                a mtl:MTLFormula ;
                mtl:hasOperator mtl:EqualTo ;
                mtl:hasRightOperand "15"^^xsd:double ;
                qudt:unit unit:knots
            ]
        ]
    ].
```

Figure 20: MTL formula representation of the ship acceleration requirement, specifying that the ship's speed must transition from above 0 knots to below 15 knots within a 300-second interval.

This formalization approach provides precise requirement representation with clear traceability between requirements and system components. The machine-readable format allows requirement reuse and modification across similar systems. Next, we examine component-level requirements, where Original Equipment Manufacturers (OEMs) define requirements bundled with their FMU components.

Component-level requirements define performance and behavioral specifications for individual powertrain elements. Original Equipment Manufacturers (OEMs) provide these requirements, which our ontology represents within Functional Mock-up Unit (FMU) definitions. Figure 21 shows an engine FMU definition with its core properties.

```
# Engine FMU definition
fmu:FMUEngine a fmu:FMU, vsso:PowertrainCombustionEngine ;
    rdfs:label "Engine FMU" ;
    fmu:fmuType "DieselEngine" ;
    fmu:FMUComponentID "ENGINE_001" ;
    fmu:hasVariable fmu:CombustionEngineMaxTorque, fmu:EngineMaxPower ;
    fmu:hasInitialParameter fmu:EngineMaxPower ;
    req:hasDefaultRequirementSet req:EngineDefaultRequirementSet .

# Variable definitions
fmu:CombustionEngineMaxTorque a fmu:FMUVariable, vsso:CombustionEngineMaxTorque ;
    fmu:hasFMUVariableName "T_ME" ;
    fmu:hasDataType xsd:double ;
    qudt:unit unit:Nm .

fmu:EngineMaxPower a fmu:FMUVariable, fmu:InitialParameter, vsso:EngineMaxPower ;
    fmu:hasFMUVariableName "engine_max_power" ;
    fmu:hasDataType xsd:double ;
    qudt:unit unit:KiloW ;
    fmu:parameterName "engine_max_power" .
```

Figure 21: Engine FMU definition showing component ID, variables, and link to default requirement set. The component ID matches its SSP definition to maintain interoperability.

Figure 22 demonstrates how requirement sets are defined using initial parameters:

```
# Conditional Default Requirement Set
req:EngineDefaultRequirementSet a req:ConditionalDefaultRequirement ;
    req:hasID "ENGINE_DEF_COND_001" ;
    req:hasDescription "Default requirements for engine based on max power" ;
    req:conditionParameter fmu:EngineMaxPower ;
    req:conditionValue "13"^^xsd:double ;
    req:hasRequirementSet req:EngineDefaultRequirementSet ;
    qudt:unit unit:MW .

# Default Requirement Set
req:EngineDefaultRequirementSet a req:ConditionalDefaultRequirement ;
    req:hasRequirement req:CombustionEngineMaxTorqueRequirement .
```

Figure 22: Engine requirement set definition showing conditional requirements based on initial parameters, such as maximum power.

Initial parameters, such as engine maximum power, determine which requirement set applies. This approach allows a single FMU to support multiple component configurations through the SSP structure. Since the SSP standard supports multiple system configurations through SSD files, requirement sets adapt to initial conditions. Figure 23 shows a specific engine requirement formalized in MTL:

```
# Requirement definition
req:CombustionEngineMaxTorqueRequirement a req:Requirement ;
    req:hasID "ENG-REQ-001" ;
    req:hasDescription "The engine shall deliver a maximum torque of at least 550 kNm at rated speed." ;
    req:hasFormula req:CombustionEngineMaxTorqueFormula ;
    req:appliesTo fmu:FMUEngine .

req:CombustionEngineMaxTorqueFormula a mtl:MTLFormula ;
    mtl:hasOperator mtl:Globally ;
    mtl:hasRightOperand [
        mtl:hasOperator mtl:True ;
            mtl:hasRightOperand [
            mtl:hasOperator mtl:LessThan ;
            mtl:hasLeftOperand fmu:CombustionEngineMaxTorque ;
            mtl:hasRightOperand [
                a mtl:MTLFormula ;
                mtl:hasOperator mtl:EqualTo ;
                mtl:hasRightOperand "550000"^^xsd:double ;
                qudt:unit unit:Nm
            ]
        ]
    ] .
```

Figure 23: Engine requirement definition specifying maximum torque limits, showing both the requirement description and its MTL formalization.

The verification is implemented on the Simantics platform [107], an open-source system for modeling and simulation. The platform provides the foundation for the MTL-based verification approach developed by Siivola [102].

4.6 Requirement Verification Implementation

The verification process consists of three main steps: simulation using OMSimulator, data collection, and requirement checking. The simulation runs the powertrain model using SSP and FMU components. The requirement verification uses the Simantics framework [107], an open-source platform for modeling and simulation of complex systems. The framework provides a Simantics Constraint Language (SCL) implementation of Metric Temporal Logic, building on the approach developed by Siivola [102]. MTL formulas derived from our ontology are transformed into the following SCL format for verification:

```
mtlNowAndForever(mtlConditionHolds(mtlLessThan(
mtlGetVariable("ship_design.Root.ENGINE_001.T_ME")::Double)
(550000.0::Double)))
```

The verification can be performed either during simulation or post-simulation. Real-time verification checks requirements at each simulation step, while post-simulation verification processes collected CSV data after simulation completion. This thesis uses the post-simulation approach to maintain consistency in data analysis across all requirements.

Requirement data for verification is extracted from the ontology using SPARQL (SPARQL Protocol and RDF Query Language). Figure 24 shows a query that retrieves requirement specifications including IDs, descriptions, formulas, and their associated components.

```
query = """
PREFIX req: <http://example.com/powertrain-requirements#>
PREFIX ssp: <http://example.com/ssp#>
PREFIX sys: <http://example.com/system#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?req ?id ?desc ?formula ?component ?componentLabel
WHERE {
    ?req a req:Requirement ;
        req:hasID ?id ;
        req:hasDescription ?desc ;
        req:hasFormula ?formula ;
        req:appliesTo ?component .
    ?component rdfs:label ?componentLabel .
}
"""
```

Figure 24: SPARQL query example for retrieving system requirements. The query uses prefixes defined in the core ontology (Section 4.2) to access requirement properties. It matches RDF triple patterns to extract requirement IDs, descriptions, formulas, and component relationships. Variables prefixed with '?' (e.g., ?req, ?id) store the matched values for output.

SPARQL queries operate on RDF graphs by matching triple patterns (subject-predicate-object relationships) against the ontology structure. This pattern matching enables precise extraction of requirement data for verification without traversing the entire ontology graph. The next section demonstrates how reasoners and competency questions verify the ontology's consistency and completeness.

4.7 Verifying Ontology Design

To verify ontology consistency and accuracy, this thesis uses the Protégé platform with the Pellet reasoner for logical verification, alongside competency questions for validation.

Protégé is an open-source tool used for ontology construction and refinement, supporting the creation of classes, properties, and relationships. The Pellet reasoner, integrated with Protégé, verifies logical consistency by detecting contradictions in the ontology's structure, such as conflicting component relationships or invalid definitions. Figure 25 shows the reasoner interface within Protégé.

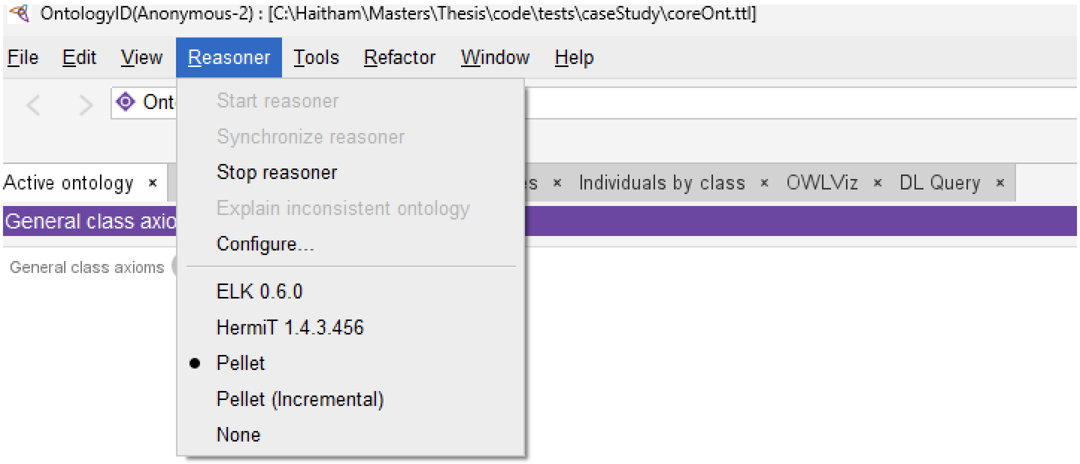

Figure 25: Pellet reasoner interface in Protégé showing classification, consistency checking, and instance verification capabilities. The reasoner employs description logic to validate ontology relationships and axioms, ensuring reliable SPARQL query execution.

Pellet interprets the ontology through description logic, evaluating axioms for logical validity and consistency. Its capabilities include classification (hierarchy organization), consistency checking (conflict detection), and instance verification (class adherence) [108]. Consistent ontology structure through Pellet verification allows reliable SPARQL query execution.

Competency questions (CQs) serve as functional benchmarks to test the ontology's ability to represent and retrieve relevant information. These questions were formulated through collaboration with expert ship designers who work with powertrain requirements and systems engineering. The selected CQs include:

1. Can the ontology represent both system-level and component-level requirements for a ship powertrain?
2. Is it possible to query the ontology to retrieve all requirements associated with a specific component, such as the engine?
3. Can the ontology express complex temporal logic formulas for dynamic requirements?
4. How effectively can the ontology represent the relationships between SSP components and their corresponding FMUs?

SPARQL queries validate the ontology against the competency questions. The following four queries demonstrate the ontology's capability to retrieve and analyze different requirement types:

Query 1 addresses competency question 1, retrieving system-level requirements:

```
PREFIX req: <http://example.com/powertrain-requirements#>
PREFIX ssp: <http://example.com/ssp#>
PREFIX sys: <http://example.com/system#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?req ?id ?desc ?formula ?component ?componentLabel
WHERE {
    ?req a req:Requirement ;
        req:hasID ?id ;
        req:hasDescription ?desc ;
        req:hasFormula ?formula ;
        req:appliesTo ?component .
    ?component rdfs:label ?componentLabel .
}
```

Query 2 extracts component default requirements:

```
PREFIX req: <http://example.com/powertrain-requirements#>
PREFIX fmu: <http://example.com/fmu#>

SELECT ?fmu ?cond_default ?id ?desc ?param ?value ?req_set
WHERE {
    ?fmu req:hasDefaultRequirementSet ?cond_default .
    ?req a req:ConditionalDefaultRequirement ;
        req:hasID ?id ;
        req:hasDescription ?desc ;
        req:conditionParameter ?param ;
        req:conditionValue ?value ;
        req:hasRequirementSet ?req_set .
}
```

Query 3 retrieves component requirement sets:

```
PREFIX req: <http://example.com/powertrain-requirements#>
PREFIX fmu: <http://example.com/fmu#>

SELECT ?cond_req ?id ?desc ?param ?paramName ?value ?req_set
WHERE {
    ?cond_req a req:ConditionalRequirement ;
        req:hasID ?id ;
        req:hasDescription ?desc ;
        req:conditionParameter ?param ;
        req:conditionValue ?value ;
        req:hasRequirementSet ?req_set .
    ?param (fmu:hasFMUVariableName|fmu:parameterName) ?paramName .
}
```

Query 4 extracts specific component requirements:

```
PREFIX req: <http://example.com/powertrain-requirements#>
SELECT ?req ?id ?desc ?formula ?component
WHERE {
    ?req_set req:hasRequirement ?req .
    ?req req:hasID ?id ;
        req:hasDescription ?desc ;
        req:hasFormula ?formula ;
        req:appliesTo ?component .
}
```

These queries demonstrate the ontology's capability to represent system-level and component-level requirements, express temporal logic formulas, and capture relationships between SSP components and FMUs.

Domain experts in ontology design and powertrain systems provided feedback to refine the ontology structure and ensure alignment with industry practices. The combination of Pellet reasoning, competency questions, and expert feedback validates both logical consistency and practical applicability of the ontology. All validation and query results are presented in Section 5. The following section describes the implementation of a ship powertrain case study, including FMU generation and SSP structure development, to test the ontology in practice.

4.8 Case Study: Powertrain System Model

To demonstrate the ontology-based approach for dynamic requirement verification, this thesis presents a ship powertrain case study. The theory for ship propulsion follows the work by MAN energy solutions [109]. The case study focuses on a ship powertrain designed for a route between Helsinki and Stockholm. Figure 26 shows the plot of speed in knots against time, serving as the test cycle for system verification.

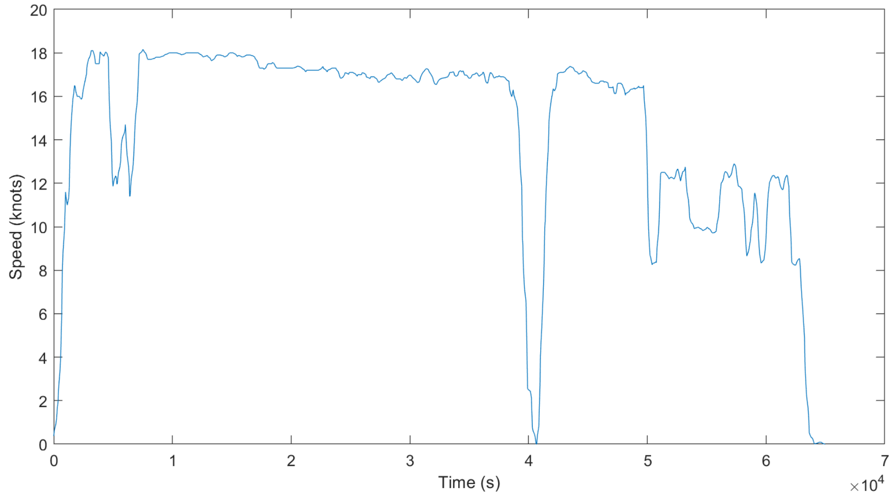

Figure 26: Journey of the ship Silja Serenade between Helsinki and Stockholm, based on the work done by Ritari, et al. [110].

The powertrain model comprises seven components: engine (main power source), controller (system operation management), hull (water interaction modeling), propeller (propulsion mechanism), consumption block (energy tracking), gearbox (power transmission), and drive cycle input (operational profile). Simulink and OpenModelica were configured to export all the models in FMI version 2.0, as described in Section 2.3.2. The SSP structure defines component interactions and OMSimulator manages simulation execution, independent of the original modeling software.

All component models are based on fundamental principles of marine propulsion from MAN's "Basic Principles of Ship Propulsion" [109]. The models incorporate relevant physical parameters and coefficients for accurate system representation. Detailed descriptions and implementations of all components, including equations and parameters, are provided in Appendix A.

The models are exported as both co-simulation and model exchange FMUs. Simulink models are exported as co-simulation FMUs using the ode4 (Runge-Kutta) solver. In co-simulation, each component solves its equations independently, with OMSimulator orchestrating data exchange between FMUs at specified communication points.

OpenModelica models are exported as model exchange FMUs using the Explicit Euler solver. The master algorithm in OMSimulator synchronizes inputs and outputs between FMUs at each time step, with configurable step sizes to balance simulation accuracy and computational performance.

4.9 SSP Model Generation

The SSP model is generated in OpenModelica, an open-source modeling and simulation environment that supports the SSP standard. OpenModelica's graphical interface enables intuitive assembly of complex systems from individual FMUs. Each component

FMU (engine, propeller, hull, and control systems) is imported and placed within the SSP structure. Figure 27 shows the SSP model created in OpenModelica for the ship powertrain simulation.

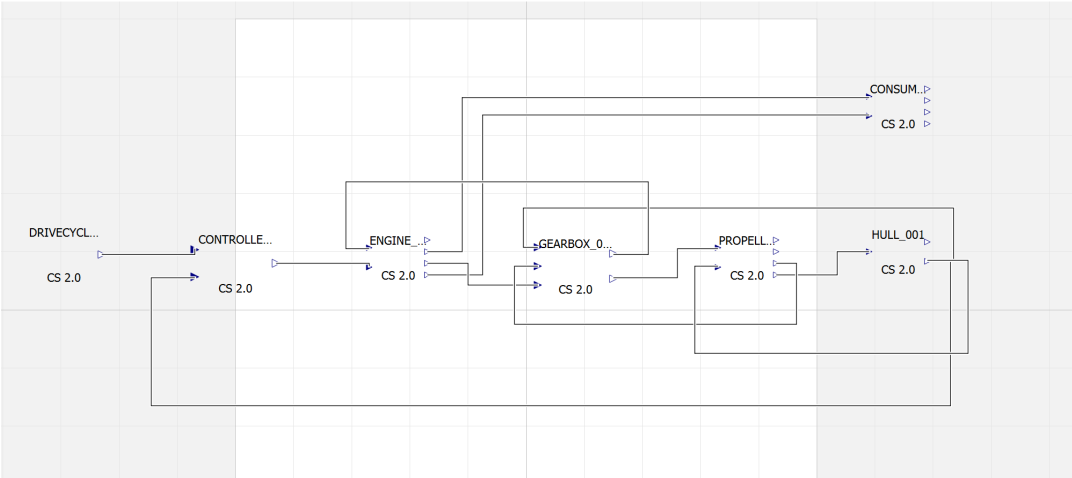

Figure 27: SSP model structure in OpenModelica showing component FMUs and their connections. The same structure was used for both model exchange and co-simulation FMUs to simulate the ship journey and collect variables for verification.

The SSP model defines connections between FMUs that represent data flow and physical interactions between ship components. OMSimulator executes the SSP model simulation and manages data exchange between FMUs according to the defined structure. All simulations were performed on a system with AMD Ryzen 7 7840HS (3.80 GHz) processor and 16GB RAM. The simulation step size was varied from 0.01s to 0.1s to analyze performance. Each configuration was run 20 times to obtain average execution times, standard deviations, maximum and minimum values, reported in Section 5.3.

This modular approach enables independent development and testing of components before system integration. The exposed variables from each FMU provide data for both system-level and component-level requirement verification, allowing comprehensive validation of the complete powertrain system.

4.10 Requirement Generation

The ontology represents requirements in a standardized, platform-independent format that can be queried using SPARQL. Since these requirements are based on MTL, they can be transformed into executable verification commands for any platform that supports temporal logic verification. This thesis uses the Simantics platform for verification, which requires a specific syntax for MTL formulas. However, the ontological representation remains independent of the verification platform - different platforms could implement their own transformations while using the same ontological source. This separation between requirement representation and verification implementation ensures flexibility and reusability of the formalized requirements.

This thesis implements eight requirements: four system-level and four component-level requirements. Each requirement is stored in a separate Turtle (.ttl) file - "ship.ttl" for system requirements and individual files (engine.ttl, propeller.ttl, etc.) for component requirements. SPARQL queries extract these requirements from the ontology, which are then converted to the verification platform's format. Figures 28 and 29 show examples of the parsed MTL formulas.

```
FMU: FMUGearbox

  Requirement Set: GEARBOX_DEF_COND_001
  Description: Default requirements for gearbox based on it engine speed
  Condition: EngineSpeed = 500.0RPM
  Default Requirement Set: GearboxDefaultRequirementSet
  Requirements:
    GEARBOX-REQ-001:
      Formula: mtlNowAndForever(mtlConditionHolds(mtlLessThan(mtlGetVariable(\"ship_design.Root.GEARBOX_001.n_ME_scaled\")
      ::Double)(477.0::Double)))
      Description: The gearbox shall be designed to operate with an input shaft speed (engine speed) not exceeding 50 rad/
      s (approximately 477 RPM) under normal operating conditions.
      Applies to:
        - FMU Component ID: GEARBOX_001, Label: Gearbox FMU
```

Figure 28: Example of a parsed MTL formula for a gearbox component requirement, showing the conversion from ontology to verification format.

```
SHIP_REQ_003:
  Formula: mtlNowAndForever(((mtlConditionHolds(mtlGreaterThan(mtlGetVariable(\"ship_design.Root.HULL_001.V_knots\")
  ::Double)(16.0::Double))) --> (mtlConditionHolds(mtlLessThan(mtlGetVariable(\"ship_design.Root.ENGINE_001.
  SSPEngineAngularVelocity_001\"):: Double)(45.0::Double)))))
  Description: If the ship speed exceeds 16 knots, then the engine angular velocity shall not exceed 45 rad/s.
  Applies to:
    - SSP Component ID: HULL_001, Label: SSP Hull
    - SSP Component ID: ENGINE_001, Label: SSP Diesel Engine
```

Figure 29: Examples of parsed MTL formulas for hull and engine system requirements, showing the conversion from ontology to verification format.

The parser maintains consistent variable naming between component-level and system-level requirements. For variables defined in both FMU and system ontologies, the parser uses the FMU-defined name. For variables only defined at the SSP level, it uses the SSP-defined name. For example, hull velocity uses its FMU name throughout, while engine angular velocity uses its SSP-defined name as it only exists at the system level.

The following examples demonstrate component-level and system-level requirements expressed in MTL and SCL:

1. Propeller Requirement (PROP-REQ-001):
   Formula:

```
(mtlConditionHolds(
   mtlGreaterThan(mtlGetVariable("ship_design.Root.PROPELLER_001.T")::Double)(
   mtlGetVariable("ship_design.Root.PROPELLER_001.T_min")::Double))
   &&
   mtlConditionHolds(mtlLessThan(
   mtlGetVariable("ship_design.Root.PROPELLER_001.T")::Double)(
   mtlGetVariable("ship_design.Root.PROPELLER_001.T_max")::Double)))
```

Description: The propeller thrust shall be 600KN, with a tolerance of ±10%.

2. Hull Requirement (HULL-REQ-001):
   Formula:

   ```
   finally (0.0, 600.0)
   (mtlConditionHolds(mtlGreaterThan(mtlGetVariable("ship_design.Root.HULL_001.
       V_knots")::Double)(15.0::Double)))
   ```

   Description: The ship's hull speed should eventually reach at least 15 knots within 10 minutes of starting.

3. Ship Requirement (SHIP_REQ_001):
   Formula:

   ```
   mtlNowAndForever(
     mtlConditionHolds(
       mtlLessThan(mtlGetVariable("ship_design.Root.HULL_001.V_knots")::Double)
           (20.0::Double)
     )
   )
   ```

   Description: The ship shall achieve a maximum speed of at least 20 knots under calm water conditions.

4. Ship Requirement (SHIP_REQ_002):
   Formula:

   ```
   mtlCausesWithinInterval(
     mtlConditionHolds(mtlGreaterThan(mtlGetVariable("ship_design.Root.HULL_001.
         V_knots")::Double)(0.0::Double)))(
     mtlConditionHolds(mtlLessThan(mtlGetVariable("ship_design.Root.HULL_001.
         V_knots")::Double)(15.0::Double))
   )(0)(300.0)
   ```

   Description: The ship powertrain system shall accelerate from 0 to 15 knots within 5 minutes.

These examples demonstrate different temporal aspects in requirements: time-independent conditions using ‘mtlNowAndForever‘, time-bounded requirements using specific intervals, and causal relationships using ‘mtlCausesWithinInterval‘.

5 Results

This chapter evaluates the ontology-based framework through multiple validation steps. Through Section 5.1, the Pellet reasoner verifies the ontology's logical consistency. Competency questions and requirement representation capabilities are demonstrated in Section 5.2. The case study results in Section 5.3 compare co-simulation and model exchange FMU performance. In Section 5.4, requirement verification results from the powertrain system implementation are presented.

5.1 Pellet Reasoner Results

The Pellet reasoner validated the ontology. It computed inferences like class hierarchy, object property hierarchy, data property hierarchy, class assertions, and property assertions. The reasoner completed these tasks without finding any logical issues, as shown in Figure 30. It processed the class hierarchy, object and data property hierarchies, and assertions in 121 milliseconds. No warnings or errors were found, indicating that the ontology is consistent.

```
INFO  17:01:00  ------------------------------ Running Reasoner ------------------------------
INFO  17:01:00  Pre-computing inferences:
INFO  17:01:00      - class hierarchy
INFO  17:01:00      - object property hierarchy
INFO  17:01:00      - data property hierarchy
INFO  17:01:00      - class assertions
INFO  17:01:00      - object property assertions
INFO  17:01:00      - data property assertions
INFO  17:01:00      - same individuals
INFO  17:01:00  Ontologies processed in 121 ms by Pellet
INFO  17:01:00
```

Figure 30: Displayed results of the pellet reasoner.

5.2 SPARQL Queries for Competency Questions

This section presents the results of SPARQL queries executed on the ontology. These results demonstrate the ontology's capability to represent and retrieve various types of requirements for powertrain systems. The queries test both system-level and component-level requirement representation capabilities.

Figure 31 shows the system-level requirements query results, demonstrating the ontology's ability to represent system-level requirements and associate them with specific components. For component-level requirements, Figure 32 shows the engine-specific query results, highlighting structured requirements with conditional elements.

```
Requirement: http://example.com/powertrain-requirements#LowPowerTemperatureRequirement
ID: MOTOR_REQ_001
Description: Motor temperature must not exceed 150C for low power motors
Formula: http://example.com/powertrain-requirements#LowPowerTemperatureMTLFormula
Component: http://example.com/fmu#FMUMotor
Component Label: FMU Motor
----------------------------------------
Requirement: http://example.com/powertrain-requirements#ComplexMotorRequirement
ID: MOTOR_REQ_002
Description: Motor temperature must not exceed 150C, max power must be between 1000W and 2000W, and these conditions must hold globally
Formula: http://example.com/powertrain-requirements#ComplexMotorMTLFormula
Component: http://example.com/fmu#FMUMotor
Component Label: FMU Motor
----------------------------------------
```

Figure 31: System-level requirements retrieved via SPARQL query, showing requirement IDs, descriptions, and component relationships. The query demonstrates successful linking between requirements and their associated system components.

```
FMU: http://example.com/fmu#FMUMotor
Default Condition: http://example.com/powertrain-requirements#MotorDefaultRequirementSet
ID: MOTOR_DEF_COND_001
Description: Default requirements for motors based on power
Parameter: http://example.com/fmu#MotorMaxPower
Value: 1500
Requirement Set: http://example.com/powertrain-requirements#MotorDefaultRequirementSet
----------------------------------------
```

Figure 32: Component-level requirements for the engine, showing conditional requirement definitions with their associated parameters and values. The query demonstrates the ontology's ability to capture complex conditional requirements.

Figure 33 presents the query results for component requirement sets. The results show the ontology can effectively represent both conditional requirement sets and individual requirements, maintaining clear component relationships.

```
Component Requirement Sets Query Results:
Conditional Requirement: http://example.com/powertrain-requirements#MotorHighPowerConditionalRequirement
ID: MOTOR_COND_REQ_002
Description: Conditional requirements for high power motor
Parameter: http://example.com/fmu#MotorMaxPower
Parameter Name: motor_max_power
Value: 2000
Requirement Set: http://example.com/powertrain-requirements#HighPowerMotorRequirementSet
----------------------------------------
Specific Component Requirements Query Results:
Requirement: http://example.com/powertrain-requirements#LowPowerTemperatureRequirement
ID: MOTOR_REQ_001
Description: Motor temperature must not exceed 150C for low power motors
Formula: http://example.com/powertrain-requirements#LowPowerTemperatureMTLFormula
Component: http://example.com/fmu#FMUMotor
----------------------------------------
```

Figure 33: Requirement sets for components showing both conditional requirement sets and individual requirements. The hierarchical structure demonstrates the ontology's capability to organize requirements at different levels of granularity.

5.3 Simulation Results in OMSimulator

The simulation of the ship powertrain model was executed using OMSimulator, with results exported to CSV files for analysis. Figure 34 shows the ship's speed using both Co-simulation and Model-Exchange FMUs.

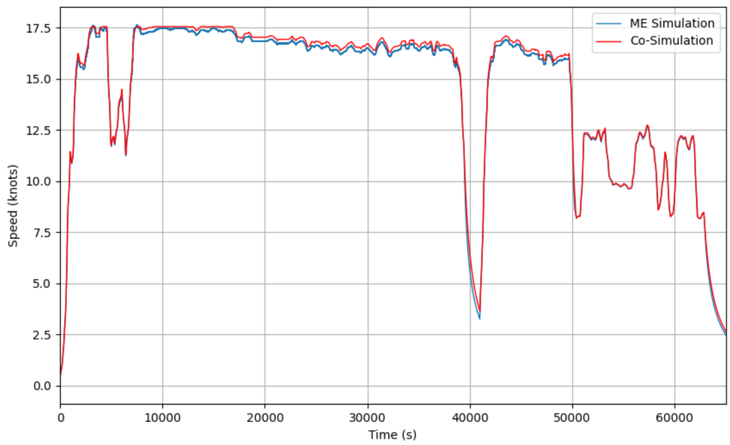

Figure 34: Ship speed in knots simulated using Co-Simulation and Model-Exchange FMUs. The differences between the implementations are discussed in Section 6.2

To evaluate simulation performance, both Co-simulation and Model Exchange FMUs were tested with varying step sizes. Each step size was tested 20 times to obtain statistical measures. Table 2 presents the results:

Table 2: Simulation times for the powertrain model using different step sizes. Each step size was measured 20 times to obtain average, maximum, minimum, and standard deviation values. Results are shown for both Co-Simulation (CS) and Model-Exchange (ME) FMUs.

Step (s)	**CS Time (s)**				**ME Time (s)**			
	Avg	**Max**	**Min**	**SD(σ)**	**Avg**	**Max**	**Min**	**SD(σ)**
0.01	120.77	126.05	116.00	2.46	60.48	62.54	59.20	0.98
0.02	61.29	63.27	58.51	1.58	31.09	33.52	29.29	1.12
0.03	40.66	45.25	39.60	1.31	22.65	28.48	19.58	3.38
0.04	31.22	33.31	30.32	0.87	15.18	15.98	14.59	0.44
0.05	26.42	27.54	25.39	0.70	11.97	12.47	11.60	0.29
0.06	21.76	22.31	21.24	0.32	10.08	10.70	9.76	0.24
0.07	19.86	21.54	18.49	0.71	8.90	10.48	8.31	0.54
0.08	16.28	16.54	16.10	0.11	7.66	7.93	7.47	0.13
0.09	14.88	15.90	14.55	0.34	6.94	8.01	6.61	0.32
0.10	13.37	13.50	13.22	0.07	6.24	6.95	6.03	0.26

Figure 35 shows how increasing the step size reduces simulation time, with the largest reduction occurring between 0.01s and 0.10s. The choice of step size balances between computational speed and accuracy, as smaller steps capture system dynamics more precisely but require longer simulation times.

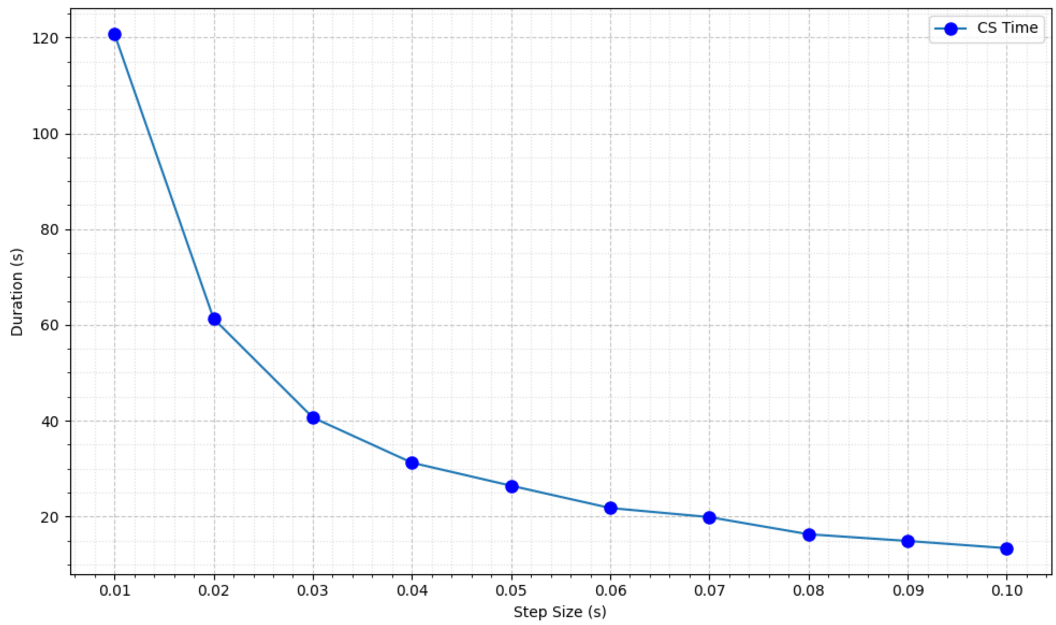

Figure 35: Average simulation time versus step size for Co-Simulation FMUs. Smaller step sizes increase simulation time but are necessary for requirements that are sensitive to rapid changes, as this affects when violations are detected.

5.4 Requirement Verification Results

The verification platform analyzed the simulation results to determine whether each requirement was met. In this context, a "violated" verdict simply indicates that a specific requirement condition was not satisfied during simulation, while "not violated" shows the requirement was fulfilled. It is important to note that a violation does not indicate a faulty component or system failure - it merely shows that the simulated behavior did not meet that particular requirement's conditions. These verdicts evaluate different design scenarios and operating conditions, allowing engineers to understand system behavior under various constraints. Table 3 presents these verification outcomes, showing each requirement's status and the time at which any violations were detected.

Table 3: The table shows requirement verification outcomes with their MTL formulas and trigger times. Trigger times indicate when a requirement violation first occurred during simulation. For requirements marked "Not Violated", no violations were detected throughout the simulation. The difference in trigger times between Co-Simulation (CS) and Model-Exchange (ME) FMUs for SHIP-003 is discussed in Section 6.3.

Name	Verdict	ID	Formula	CS (s)	ME (s)
hull1	Violated	HULL-001	$\Diamond_{[0,600]}(V > 15)$	600.0	600.0
cons1	Violated	CONS-001	$\Box_{[290,330]}(P_f < 800)$	290.01	290.01
eng1	Not Violated	ENG-001	$\Box(T < 550000)$	–	–
gear1	Not Violated	GEAR-001	$\Box(n < 477)$	–	–
prop1	Violated	PROP-001	$\Box(T_{\text{min}} < T < T_{\text{max}})$	0.0	0.0
sys1	Not Violated	SHIP-001	$\Box(V < 20)$	–	–
sys2	Violated	SHIP-002	$(V > 0) \rightsquigarrow_{[0,300]} (V < 15)$	300.0	300.0
sys3	Violated	SHIP-003	$\Box(V > 16 \rightarrow \omega < 45)$	2660.07	2751.6

The verification platform correctly identified requirement violations by comparing simulation data against the specified conditions. The accuracy of these verdicts was confirmed by examining the CSV data files at the reported trigger times. For example, at t=600s, the hull velocity data confirms that HULL-001's requirement (reaching 15 knots within 10 minutes) was indeed not met, validating the platform's violation detection. Similarly, all other verdicts were verified through time-series analysis of the relevant variables, demonstrating that the verification system accurately assessed requirement compliance.

To assess the efficiency of the verification platform, the parsing time is analyzed for each CSV files with different time steps, using the same hardware setup described in Section 4.9 (AMD Ryzen 7 7840HS, 16GB RAM). Each step size was tested 20 times to obtain statistical measures. Table 4 presents the results:

Table 4: CSV file parsing times for different step sizes. Each step size was measured 20 times to obtain average, maximum, minimum, and standard deviation values. Results are shown for both Co-Simulation (CS) and Model-Exchange (ME) FMUs.

Step (s)	CS Time (s)				ME Time (s)			
	Avg	Max	Min	SD(σ)	Avg	Max	Min	SD(σ)
0.01	71.2	73.1	69.8	1.2	84.7	86.9	82.9	1.5
0.02	55.5	57.2	54.1	0.9	70.6	72.4	69.1	1.2
0.03	33.8	35.1	32.7	0.7	45.3	46.8	44.1	0.9
0.04	25.2	26.3	24.4	0.6	36.4	37.8	35.2	0.8
0.05	21.4	22.3	20.7	0.5	30.2	31.4	29.3	0.7
0.06	17.4	18.2	16.8	0.4	26.5	27.6	25.7	0.6
0.07	14.7	15.4	14.2	0.4	22.1	23.1	21.3	0.5
0.08	13.0	13.6	12.5	0.3	19.8	20.7	19.1	0.5
0.09	12.0	12.5	11.6	0.3	17.9	18.7	17.3	0.4
0.10	11.0	11.4	10.7	0.2	17.2	17.9	16.6	0.4

Figure 36 shows that parsing time decreases with larger step sizes, as fewer data points need processing. Model-Exchange FMUs consistently require longer parsing times due to additional variable aliases in the CSV files.

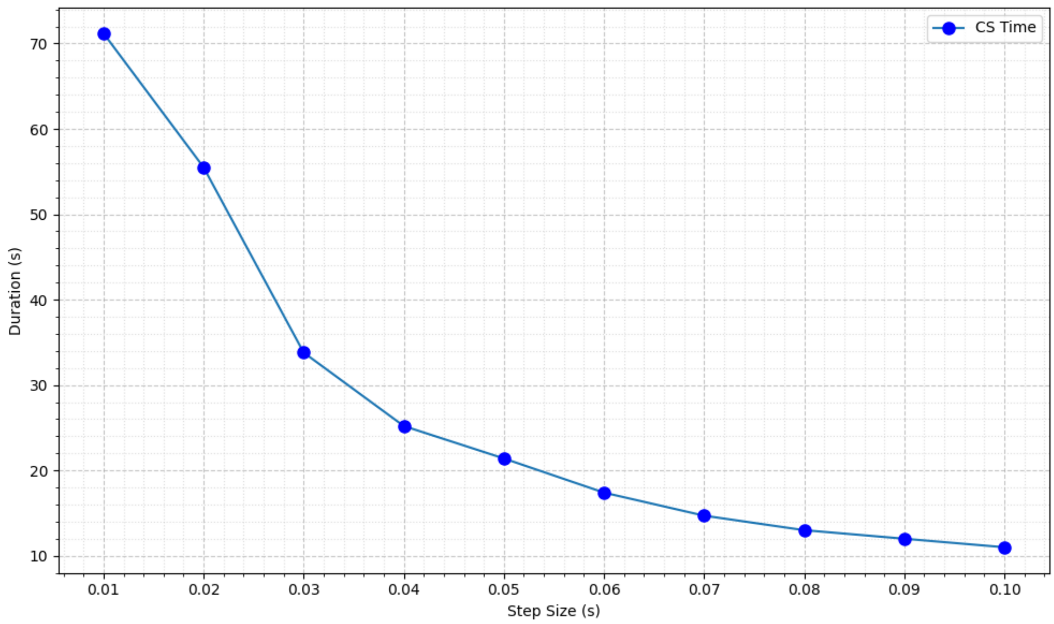

Figure 36: Average parsing time versus step size for Co-Simulation FMUs. Smaller step sizes increase parsing time as more data points need to be processed for requirement verification.

This chapter evaluated the ontology through four main aspects: logical consistency using the Pellet reasoner, completeness through competency questions, practical implementation in a ship powertrain case study using FMI and SSP standards, and correctness of formalized dynamic requirements through requirement verification. Section 6 analyzes these results in relation to the research gaps identified in Section 3.6.

6 Discussion

This chapter discusses the performance and capabilities of the ontology-based framework for dynamic requirement verification in powertrain systems. Section 6.1 evaluates the ontology's capability to represent and verify requirements through reasoner results and competency questions. The simulation performance of co-simulation and model exchange FMUs in the ship powertrain case study is examined in Section 6.2. Section 6.3 assesses the verification process and MTL-based requirement verification. Finally, Section 6.4 discusses limitations of the current approach and proposes directions for future research.

6.1 Ontology Evaluation

This thesis contributes to systems engineering through providing a structured, machine-readable approach to dynamic requirement verification. The scientific contribution bridges the gap between ontological knowledge representation and temporal logic verification - previous approaches handled only static requirements or simple temporal constraints. In large-scale engineering projects involving thousands of requirements across multiple components, manual verification becomes impractical. This ontology-based approach addresses this through formal representation of dynamic requirements using Metric Temporal Logic, enabling precise specification and systematic verification. The evaluation of the developed ontology examines the Pellet reasoner results, SPARQL query outcomes, and ontology completeness through multiple validation methods.

As shown in Section 5.1, the Pellet reasoner validated the ontology's logical consistency using methods described in Section 2.4.1. The reasoner processed class hierarchies, property relationships, and assertions in 121 milliseconds, detecting no logical issues (Figure 30). This validation confirms the ontology's internal consistency and freedom from contradictions. The Resource Description Framework (RDF) model employs two types of properties: object properties connecting subject-predicate-object triplets, and datatype properties linking individuals to literal values.

These validation results address part of the research gap identified in Section 3.6 concerning rigorous methods for dynamic requirement representation. Unlike previous work, this approach handles temporal requirements while maintaining integration with industry-standard simulation tools, through querying the ontology. The ontology formalizes temporal requirements using MTL concepts, enabling systematic querying and reuse across different systems.

Competency Questions (CQs) validate the ontology's capability to represent dynamic requirements. The SPARQL queries, detailed in Section 5.2, successfully retrieved both system-level (Figure 31) and component-level requirements (Figure 32). The ontology represents and queries different requirement types, including conditional requirements and requirement sets (Figure 33) - an improvement over existing frameworks that handle only static requirements.

This capability addresses the research gap in integrating ontologies with formal methods for requirement specification and verification. The approach combines ontological semantics with the ability to specify and query dynamic system properties, beyond static requirement representation in existing frameworks.

The ontology's completeness is demonstrated through its representation of powertrain systems and their requirements, validated through CQs. The successful retrieval of system-level, component-level, and conditional requirements through SPARQL queries indicates comprehensive domain coverage. It is important to note that although the ontology addresses many of the gaps identified in Section 3.6, there is room for improvement. For instance, the challenge of efficiently updating and verifying requirements during the system lifecycle, which is characteristic of MBSE, is not addressed in the current approach, which has no way of updating the requirements as system development progresses. Additionally, our approach is able to translate requirements to dynamic and verifiable commands; it still lacks a structured approach to translating the requirements to their logical form from their initial natural language formalization.

Qualitative engineering terms challenge logical representation. "Steady-state" conditions, defined as system variables remaining constant over time, become complex when translated to MTL structures. Such conditions often require multiple formulas to represent parameter stability, resulting in verbose representations.

Similarly, certain powertrain concepts resist logical formalization. Component wear and fluid dynamics involve continuous, non-linear behaviors that challenge discrete logical representation. The extensible nature of ontologies provides flexibility in addressing these limitations. This thesis leverages terms from the Vehicle Signal Specification Ontology (VSSo) and Semantic Sensor Network Ontology (SOSA) for ship powertrain components - demonstrating how domain-specific ontologies enhance representation capabilities while maintaining compatibility with existing engineering tools and workflows.

The approach provides a practical solution for collaborative design environments, where requirement complexity increases with system scale. By enabling verification of temporal requirements, it reduces verification time and improves accuracy compared to manual methods, directly addressing the needs of modern engineering projects.

Unlike traditional methods that primarily focus on static requirements or simple temporal constraints, this research integrates ontological frameworks with Metric Temporal Logic (MTL) to enable dynamic requirement verification. Previous works, such as Cleland-Huang et al. [18] and Sun et al. [10], were limited to static requirement modeling using SysML and UML frameworks. By leveraging the flexibility of ontologies and the precision of MTL, this approach allows for the formal representation of complex temporal dependencies, which previous frameworks could not effectively address. This enhancement reduces verification time and improves accuracy, particularly in large-scale systems with thousands of interconnected requirements.

6.2 Case Study Simulation Results

This case study demonstrates the application of our ontology-based verification approach in a multi-tool, co-simulation environment. The implementation combines different modeling tools and simulation methods to verify dynamic requirements in powertrain systems - addressing practical engineering needs where teams work with software platforms.

The simulation of the ship powertrain model, described in Section 4.8, was executed using OMSimulator. The ship models were developed in both Simulink and OpenModelica and exported as FMUs, using the SSP structure to construct the ship system. The use of different tools to export different models demonstrates the interoperability of the Functional Mock-up Interface (FMI) and System Structure and Parameterization (SSP) standards through their tool-agnostic nature.

The initial models were created in Simulink and exported for co-simulation, with each FMU containing its own solver. In co-simulation, each component solves its equations independently, with OMSimulator managing the data exchange between FMUs. For model exchange, identical models were created in OpenModelica and exported as model exchange FMUs using the Explicit Euler solver. Since the models have dependent variables across each FMU, algebraic loops and initial conditions can cause solver convergence failures. This problem occurs with model exchange FMUs because the same master algorithm handles both synchronization and differential equation solving within each model. Approximations were necessary to smooth signals between FMUs. For example, the propeller's rotational speed is described by a differential equation:

$$\alpha = \frac{T - Q}{\text{Inertia}}, \tag{8}$$

where T and Q are the torques of the engine and propeller, and α is the angular acceleration.

$$\omega = \int \alpha \, dt, \tag{9}$$

where ω is the rotational speed obtained by integrating α. The direct connection of rotational speed output to the ship engine FMU creates a potential algebraic loop. In model exchange, the master algorithm solves these differential equations at each time step. Larger step sizes can lead to significant numerical instability. OpenModelica provides specific functions to smooth signals and ensure equation differentiability, though this approach creates minor differences between OpenModelica and Simulink models. While these differences did not cause significant divergence in system dynamics, Section 6.3 discusses the observed variations.

Table 2 and Figure 34 show the ship's output speed for both co-simulation and model exchange FMUs. The dynamics remain nearly identical, though model exchange FMUs track drive cycle fluctuations more precisely. This stems from how OMSimulator handles data exchange. Co-simulation FMUs use separate solvers that exchange data at fixed intervals, smoothing small fluctuations. Model exchange FMUs use a single solver - this enables tighter integration but creates delays between FMUs. Co-simulation shows smaller input-output differences through interpolation, while model exchange shows larger differences due to its unified solving approach.

Table 2 and Figure 35 present simulation times across different step sizes. Simulation duration decreases with increasing step size for both FMU types, with model exchange FMUs performing faster. At 0.01s step size, co-simulation requires 120.77s compared to model exchange at 60.48s. This performance difference arises because co-simulation involves both internal FMU solving and inter-FMU synchronization, while model exchange only solves the complete system at each step.

This approach demonstrates tool-agnostic verification in complex systems through successful integration of both co-simulation and model exchange modeling methods. Our verification across different modeling environments addresses practical industry needs. Teams using different software tools benefit from reduced development time, streamlined collaboration, and more reliable requirement verification.

Uncertainty in the results stems primarily from numerical approximations in signal smoothing and solver interactions. These uncertainties did not significantly affect requirement verification outcomes, suggesting the approach is robust enough for practical engineering applications. However, numerical instability, particularly in model exchange FMUs, can arise from algebraic loops and differential equation solving. Larger step sizes, while reducing computation time, may introduce errors due to reduced accuracy in variable synchronization.

Future research could explore adaptive step-size algorithms to optimize the trade-off between accuracy and computation time, especially in systems with widely varying time constants. Additionally, solver optimization techniques in tools like OpenModelica and Simulink could further mitigate numerical instability. By refining solver interactions and exploring signal smoothing techniques, the accuracy of the verification process can be improved, ensuring more reliable outcomes even in highly complex simulations.

6.3 Requirement Verification

The requirement verifications were performed as described in Section 4.6. This platform evaluates whether the simulated system behavior meets the specified requirements, with the outcomes shown in Table 3.

The verification results indicate if and when requirements were violated during simulation. A "violated" verdict simply shows that a specific requirement condition was not satisfied, while "not violated" indicates the requirement was fulfilled. These verdicts do not indicate faulty components or system failures - they show whether simulated behavior met the specified conditions. This distinction matters for evaluating different design scenarios and understanding system behavior under various constraints.

This systematic requirement verification supports modern engineering projects where multiple stakeholders collaborate. System integrators can verify requirements from different component manufacturers without detailed knowledge of internal component designs. Component manufacturers can test their products against system-level requirements before physical integration. The formalized requirements provide a common reference between teams, reducing misunderstandings and enabling early detection of integration issues.

Both co-simulation and model exchange FMUs produced similar violation results at nearly identical times, except for one system requirement (SHIP_REQ_003) that triggered slightly later in the model exchange simulation. This difference is due to the data smoothing applied in model exchange FMUs to avoid numerical instability which changed the response signal resulting in the requirement violation being triggered 91.53 seconds later.

Table 4 shows the time required to parse the simulation results. The time taken to parse the data was longer for model exchange FMUs compared to co-simulation. Although both methods produced the same number of steps for the drive cycle, the model exchange files contained more columns of data due to multiple variable aliases. These aliases, used for block synchronization within the SSP structure, increase parsing time. However, SSP's signal filtering feature can mitigate this issue by selecting specific data for collection. Figure 36 shows the difference in parsing times for co-simulation FMUs. The differences in how data is handled and the choice of solver can change the performance of the simulation and the verification process.

The verification framework's ability to work with different simulation approaches and handle complex requirements positions it well for integration with emerging technologies. Digital twins, IoT systems, and advanced simulation platforms can benefit from this structured approach to requirement verification, particularly in scenarios where continuous verification of system behavior against specified requirements is necessary.

6.4 Limitations of the Current Approach and Future Work

This thesis advances systems engineering through the formalization and verification of dynamic requirements. The scientific impact lies in bridging ontological knowledge representation with temporal logic verification - allowing for requirement verification in complex systems. With this approach, requirements with temporal constraints can be formally represented and verified across different simulation environments. This contribution addresses a fundamental challenge in systems engineering where traditional approaches struggled to verify time-dependent behaviors.

The practical impact manifests in large-scale engineering projects involving multiple stakeholders. These projects often contain thousands of requirements spanning numerous components with complex temporal relationships. Our approach enables engineers to formally specify these requirements, verify them through simulation, and maintain traceability between system-level and component-level specifications. This capability directly supports modern development practices where different teams collaborate using varied tools and methodologies.

The approach integrates System Structure and Parameterization (SSP), Functional Mock-up Interface (FMI) and Metric Temporal Logic (MTL) to verify dynamic requirements. However, several limitations exist in the current implementation. First, requirements must be expressible within simulation models and mappable to MTL formulas in the ontology. Requirements involving complex physical phenomena or continuous behaviors often require significant expertise to translate into verifiable forms. The verification accuracy depends heavily on simulation model fidelity and requires domain expertise to interpret results in the context of real-world systems.

Formal verification tools like Temporal Logic of Actions Plus (TLA+) offer complementary capabilities by verifying all possible system states. Kibret et al. [90] demonstrated how category theory could formalize relationships between different system representations. This theoretical foundation suggests several concrete research directions. Future work should develop formal methods to verify requirements without simulation, potentially through combinations of theorem proving and model checking. Research could focus on creating automated translators between natural language requirements and MTL formulas. Machine learning techniques could improve requirement formalization from historical data, while extensions to the ontology could capture physical phenomena currently difficult to represent.

The current implementation also lacks mechanisms for requirement evolution during system development. Research by Lu et al. [88] on automatic updates between models and ontologies provides insight into managing evolving specifications. An approach to requirement evolution needs to address how requirements change throughout a system's lifecycle. This includes maintaining consistency when propagating changes across system hierarchies, implementing version control for requirements, and integrating with product lifecycle management systems to track these changes.

Future research should explore expanding the ontology to handle different types of requirements. For example, requirements about continuous physical wear could use different formalisms than discrete temporal requirements. The ontology could be extended to incorporate additional logical frameworks beyond MTL for requirements that cannot be verified through simulation alone. For instance, safety-critical requirements might benefit from theorem-proving approaches, while performance requirements could continue using simulation-based verification.

Building on these ontological expansions, another promising direction for future research is the development of a standardized repository for Digital Twin (DT) prototypes that includes their formalized requirements. Current digital twin prototype implementations offer model sharing capabilities but lack standardized methods to communicate associated requirements. Building on the ontology-based approach presented in this thesis, future work could develop a comprehensive database that stores not only virtual models but also their formal requirements, verification histories, and associated metadata.

Such a repository would enable engineers to access complete digital twin prototype packages including verified requirement specifications and reuse established requirement patterns across similar systems. System designers could combine these embedded component specifications with their own system-level requirements, creating complete verification frameworks. The repository could track requirement evolution across different versions of digital twin prototypes, building a knowledge base of common requirement formalization patterns in specific domains. This would advance the field beyond simple model exchange toward complete system knowledge sharing, aligning with the core concept of digital twins as comprehensive virtual representations. Moreover, it would extend their utility from pure simulation to include formal requirement specifications and verification results.

7 Conclusion

This thesis developed an ontology-based approach for formalizing and verifying dynamic requirements in powertrain systems. The research bridged a critical gap between ontological knowledge representation and temporal logic verification through three main contributions. First, it created an ontology that represents Metric Temporal Logic (MTL) formulas, enabling precise specification of time-dependent requirements. Second, it demonstrated integration with System Structure and Parameterization (SSP) and Functional Mock-up Interface (FMI) standards, showing how formalized requirements can work within existing simulation platforms. Third, it provided a systematic method for verifying these requirements through simulation-based testing.

This research validated the approach implemented through a ship powertrain case study, implementing both co-simulation and model exchange Functional Mock-up Units (FMUs). Performance analysis revealed that model exchange FMUs achieved faster simulation times but required additional data processing during verification. The ontology successfully captured and verified complex requirements spanning multiple components, with the requirement verification effectively detecting requirement breaches in both real-time and post-simulation scenarios.

Modern engineering projects are becoming increasingly complex, integrating components from diverse manufacturers, each with their unique requirements and specifications. The ability to verify these requirements early in the design process is critical for preventing costly integration issues and reducing development time. This thesis advances systems engineering by developing an ontology-based approach for specifying and verifying dynamic requirements before physical prototyping. This method ensures that stakeholders can collaborate with a unified understanding of system requirements, addressing gaps in early-stage verification.

The maritime industry has already seen benefits from applying this approach to ship powertrain design, where early requirement verification facilitates the seamless integration of engines, propulsion systems, and control units. Beyond the maritime sector, this ontology-based framework reduces design verification times and enhances system reliability by enabling early detection of requirement breaches. For instance, in the automotive industry, this approach could streamline the integration of electronic control units (ECUs) from different suppliers, reducing development costs and time-to-market. As modern engineering systems continue to grow in complexity, managing hundreds or even thousands of dynamic requirements, the need for formal requirement verification is becoming not just beneficial but essential.

This thesis contributes to the field by integrating an ontology-based framework with co-simulation tools to enable real-time requirement verification. Future research can build on top of this work by extending the ontology framework to encompass even more intricate multi-domain systems, such as integrated aerospace platforms or autonomous vehicle networks, further pushing the boundaries of early-stage requirement verification. By expanding the ontology framework and incorporating additional logical methods, this research lays a foundation for more efficient, collaborative design processes, driving innovation and sustainability in complex engineering systems.

A Ship powertrain model

This appendix provides some details of the ship powertrain components. The mathematical formulae and assumption are based on the principles of ship propulsion taken from [109]. Examples of the Hull and Propeller models are shown here.

A.1 Propeller Model

The propeller model implements wake fraction calculations, thrust coefficients, and torque dynamics with smooth approximations for numerical stability.

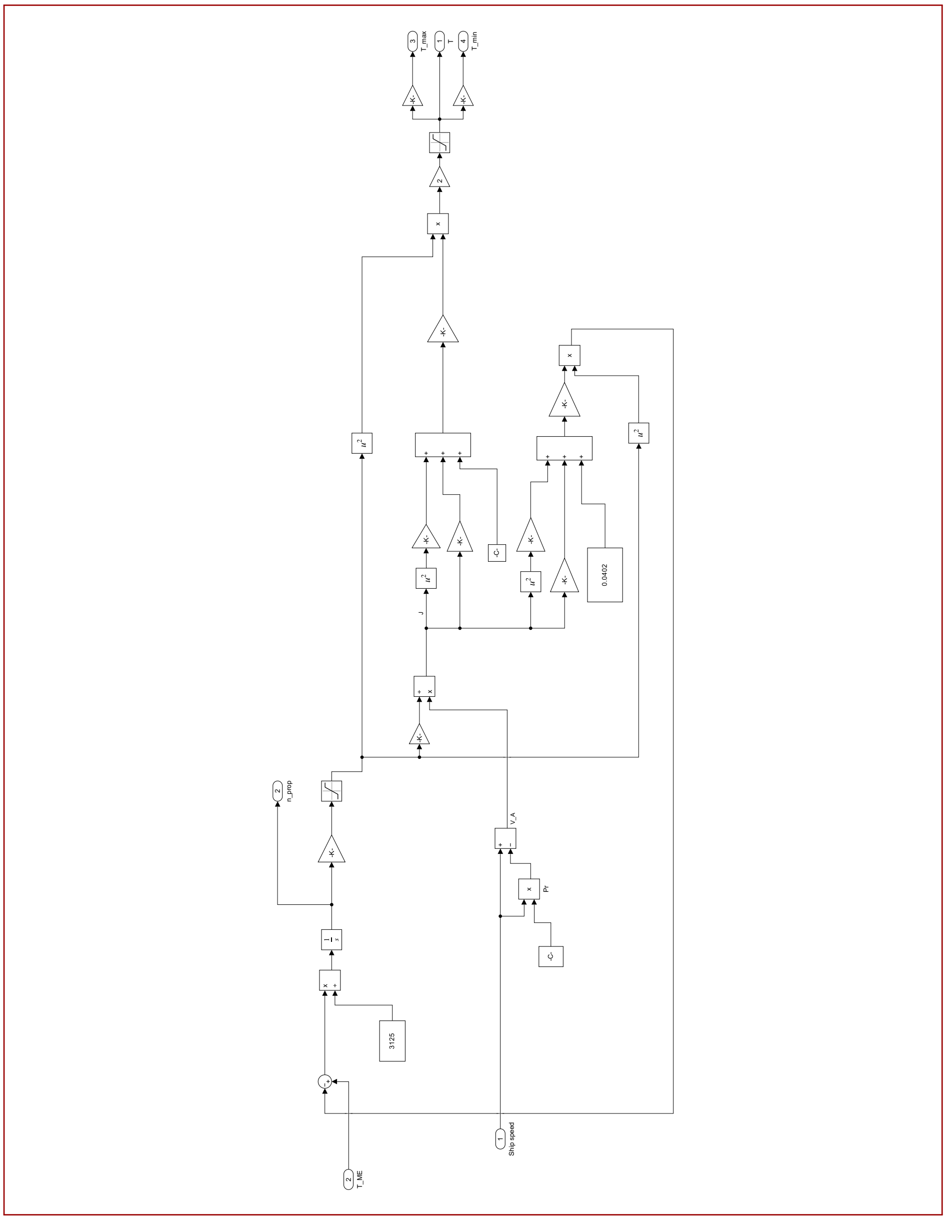

Figure A1: Block diagram of the propeller model showing signal flows and calculations.

```
model PROPELLER_001
  input Real ship_speed "Ship speed";
  input Real T_ME "Engine torque";

  // Outputs
  output Real T(start=0) "Thrust";
  output Real n_prop(start=0.1);
  output Real T_max "Max thrust (N)";
  output Real T_min "Min thrust (N)";

  // Parameters
  parameter Real alpha = 3125 "Moment of inertia of the prop";
  parameter Real w = 0.1 "wake fraction";
  parameter Real D = 5 "diameter of prop";
  parameter Real rho = 1025 "rho";
  parameter Real epsilon = 0.1;  // Increased to avoid division by very small numbers

  // Variables
  Real omega(start=0.1);
  Real V_a(start=0);
  Real Q(start=0);
  Real J(start=0.1);
  Real K_Q(start=0.0402);
  Real K_T(start=0.3486);

  // Smooth approximation of max function
  function smooth_max
    input Real x;
    input Real y;
    input Real k;
    output Real result;
  algorithm
    result := 0.5 * (x + y + sqrt((x - y)^2 + k));
  end smooth_max;

equation
  V_a = ship_speed * (1 - w);

  // Modified differential equation to reduce stiffness
  der(omega) = smooth_max(0, (T_ME - Q) / alpha, 1e-3);

  n_prop = omega * 0.15915;

  // Modified J calculation to avoid division by very small numbers
  J = smooth_max(epsilon, V_a, 1e-3) / smooth_max(epsilon, n_prop * D, 1e-3);

  // Smoothed versions of K_Q and K_T calculations
  K_Q = smooth_max(0, -(0.0124 * J^2) - (0.0311 * J) + 0.0402, 1e-6);
  K_T = smooth_max(0, -(0.1429 * J^2) - (0.3286 * J) + 0.3486, 1e-6);

  Q = K_Q * rho * n_prop^2 * D^5;
  T = K_T * rho * n_prop^2 * D^4 * 2;

  T_max = T * 1.1;
  T_min = T * 0.9;
end PROPELLER_001;
```

Listing 2: OpenModelica implementation of the propeller model.

A.2 Hull Model

The hull model implements resistance calculations based on Froude number interpolation, incorporating mass dynamics and velocity conversions. The resistance coefficients are derived from experimental data and interpolated based on the current ship speed.

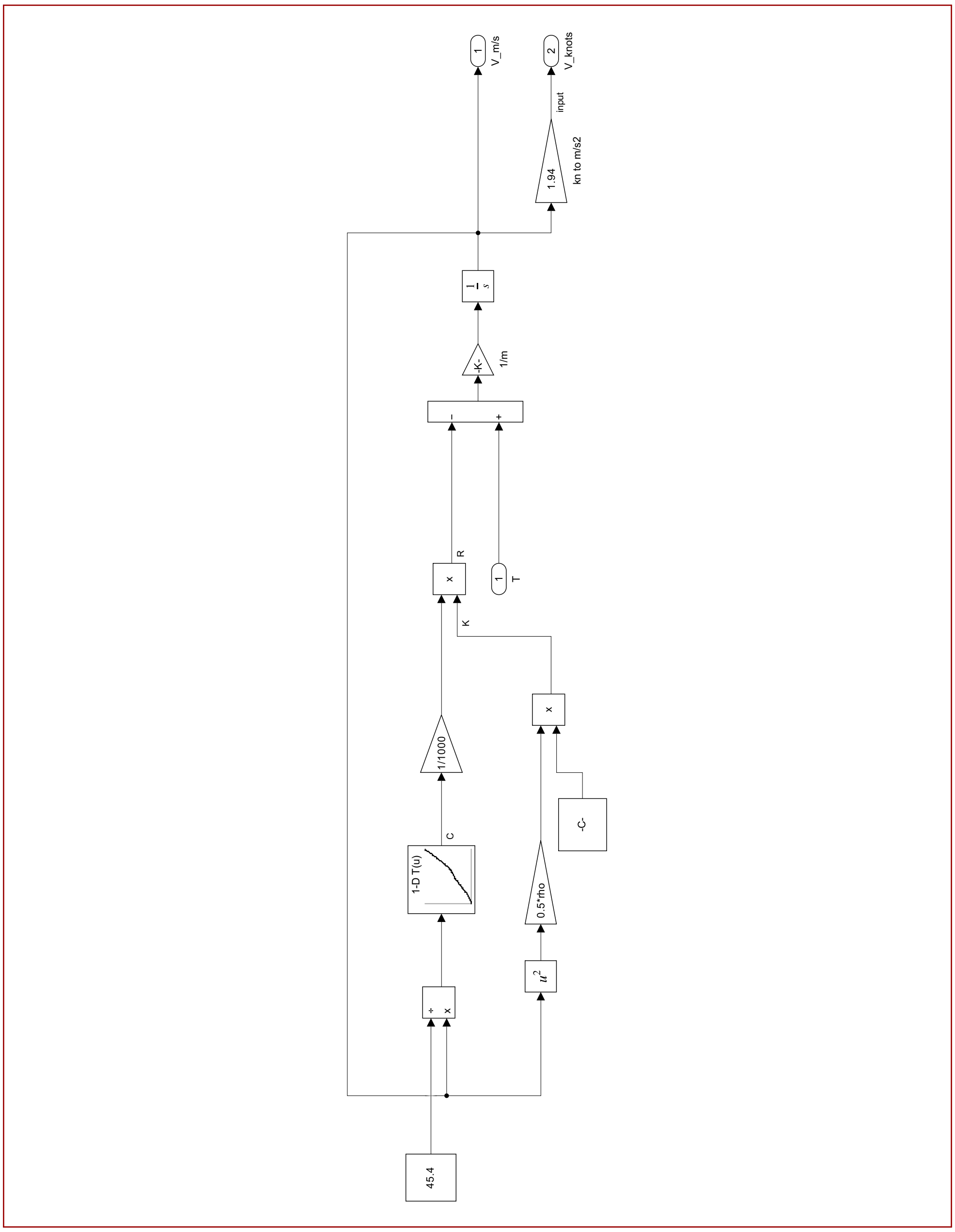

Figure A2: Block diagram of the hull model showing resistance calculation, force dynamics, and velocity outputs.

```
model HULL_001
  parameter Real mass = 32000000 "mass of the hull in kg";
  parameter Real resis_coeff[75] = {2.343949, 2.363057, 2.372611, 2.391720, 2.410828,
    2.429936, 2.458599, 2.487261, 2.496815, 2.525478, 2.544586, 2.563694, 2.592357,
    2.621019, 2.659236, 2.687898, 2.726115, 2.773885, 2.802548, 2.831210, 2.859873,
    2.898089, 2.926752, 2.974522, 3.003185, 3.050955, 3.079618, 3.108280, 3.136943,
    3.165605, 3.194268, 3.222930, 3.232484, 3.270701, 3.289809, 3.299363, 3.318471,
    3.347134, 3.375796, 3.404459, 3.433121, 3.461783, 3.490446, 3.528662, 3.557325,
    3.595541, 3.643312, 3.691083, 3.729299, 3.757962, 3.805732, 3.853503, 3.901274,
    3.939490, 3.996815, 4.025478, 4.082803, 4.111465, 4.140127, 4.178344, 4.226115,
    4.264331, 4.321656, 4.359873, 4.407643, 4.445860, 4.493631, 4.531847, 4.579618,
    4.627389, 4.675159, 4.722930, 4.770701, 4.808917, 4.847134};
  parameter Real frd_num[75] = {0.190000, 0.191429, 0.192857, 0.194286, 0.195714,
    0.196905, 0.198571, 0.199762, 0.200952, 0.201905, 0.202857, 0.204048, 0.205238,
```

```
    0.206667, 0.208095, 0.209286, 0.210476, 0.211667, 0.213095, 0.214048, 0.215000,
    0.216429, 0.217619, 0.218810, 0.220238, 0.221667, 0.222857, 0.223810, 0.225000,
    0.226429, 0.227381, 0.228333, 0.229762, 0.230952, 0.231905, 0.233095, 0.234048,
    0.235238, 0.236667, 0.237619, 0.239524, 0.240476, 0.241429, 0.242381, 0.243571,
    0.244762, 0.245714, 0.246905, 0.247857, 0.248810, 0.249762, 0.250952, 0.251905,
    0.252857, 0.253810, 0.254762, 0.255476, 0.255952, 0.256667, 0.257857, 0.258810,
    0.259762, 0.260714, 0.261667, 0.262381, 0.263571, 0.264286, 0.265000, 0.265952,
    0.266905, 0.267857, 0.268571, 0.269524, 0.270238, 0.270952};

  input Real T "thrust input in Newtons";

  // Outputs
  output Real V_knots "V in knots";
  output Real V(start=0) "V in m/s";

  Real R(start=0);
  Real a(start=0) "acceleration";
  Real FroudeNumber(start=0) "froude number for lookup";
  Real resis_coeff_interp(start=2.343949) "interpolated resistance coefficient";

equation
  FroudeNumber =  V / 45.4;

  // interpolation
  resis_coeff_interp = Modelica.Math.Vectors.interpolate(
    frd_num,
    resis_coeff,
    FroudeNumber
  );

  // calculate resistance R
  R = (resis_coeff_interp / 1000) * (515 * V^2 * 6187.9);

  // f = ma
  a = (T - R) / mass;

  // integrates
  der(V) = a;

  // v in knots
  V_knots = V * 1.94;

end HULL_001;
```

Listing 3: OpenModelica implementation of the hull model.

B Further Requirement Examples

This appendix provides additional examples of requirement formalization using the ontology-based approach. The following example shows a ship requirement that relates speed to engine angular velocity, demonstrating both complex requirement definitions and variable handling through multiple ontologies (VSSo, SOSA, and SSN):

```
req:ShipSpeedAngularVelocityRequirement a req:Requirement ;
    req:hasID "SHIP_REQ_003" ;
    req:hasDescription "If the ship speed exceeds 16 knots, then the engine angular
        velocity shall not exceed 45 rad/s." ;
    req:hasFormula req:ShipSpeedAngularVelocityFormula ;
    req:appliesTo sys:SSPHull, sys:SSPDieselEngine .

req:ShipSpeedAngularVelocityFormula a mtl:MTLFormula ;
    mtl:hasOperator mtl:Globally ;
    mtl:hasRightOperand [
        a mtl:MTLFormula ;
        mtl:hasOperator mtl:IfThen ;
        mtl:hasLeftOperand [
            mtl:hasOperator mtl:True ;
            mtl:hasRightOperand [
                a mtl:MTLFormula ;
                mtl:hasOperator mtl:GreaterThan ;
                mtl:hasLeftOperand sys:SSPHullSpeed ;
                mtl:hasRightOperand [
                    a mtl:MTLFormula ;
                    mtl:hasOperator mtl:EqualTo ;
                    mtl:hasRightOperand "16"^^xsd:double ;
                    qudt:unit unit:Knot
                ]
            ]
        ] ;
        mtl:hasRightOperand [
            mtl:hasOperator mtl:True ;
            mtl:hasRightOperand [
                a mtl:MTLFormula ;
                mtl:hasOperator mtl:LessThan ;
                mtl:hasLeftOperand sys:SSPEngineAngularVelocity_001 ;
                mtl:hasRightOperand [
                    a mtl:MTLFormula ;
                    mtl:hasOperator mtl:EqualTo ;
                    mtl:hasRightOperand "45"^^xsd:double ;
                    qudt:unit unit:rad
                ]
            ]
        ]
    ] .
```

Listing 4: Ontological definition of a ship speed and angular velocity requirement.

The parser transforms this ontological representation into an executable MTL formula:

```
SHIP_REQ_003:
```

```
  Formula: mtlNowAndForever(((mtlConditionHolds(mtlGreaterThan(
    mtlGetVariable("ship_design.Root.HULL_001.V_knots")::Double)(16.0::Double)))
    --> (mtlConditionHolds(mtlLessThan(
    mtlGetVariable("ship_design.Root.ENGINE_001.SSPEngineAngularVelocity_001")::Double)
    (45.0::Double)))))
  Description: If the ship speed exceeds 16 knots, then the engine angular
    velocity shall not exceed 45 rad/s.
  Applies to:
    - SSP Component ID: HULL_001, Label: SSP Hull
    - SSP Component ID: ENGINE_001, Label: SSP Diesel Engine
```

Listing 5: Parsed MTL formula for the speed-angular velocity requirement.

The ontology uses a hierarchical approach to variable definition, starting with VSSo terms and falling back to SOSA/SSN ontologies when needed. The following example demonstrates this strategy:

```
fmu:FMUMotor a fmu:FMU, vsso:PowertrainElectricMotor ;
    rdfs:label "FMU Motor" ;
    fmu:fmuType "ElectricMotor" ;
    fmu:FMUComponentID "MOTOR_001" ;
    fmu:hasVariable fmu:MotorTemperature, fmu:MotorMaxPower ;
    fmu:hasInitialParameter fmu:MotorMaxPower ;
    req:hasDefaultRequirementSet req:MotorDefaultRequirementSet .

# Using VSSo ontology
fmu:MotorTemperature a fmu:FMUVariable, vsso:MotorTemperature ;
    fmu:hasFMUVariableName "motor_temperature" ;
    fmu:hasDataType xsd:double ;
    qudt:unit unit:DEG_C .

# Using SOSA/SSN for undefined variables
fmu:MotorWindingTemperature a fmu:FMUVariable, sosa:ObservableProperty ;
    fmu:hasFMUVariableName "motor_winding_temperature" ;
    fmu:hasDataType xsd:double ;
    qudt:unit unit:DEG_C ;
    ssn:isPropertyOf fmu:MotorTemperature .
```

Listing 6: Variable definition using VSSo and SOSA/SSN ontologies.

The example also demonstrates how initial parameters are used to define default requirement sets:

```
fmu:MotorMaxPower a fmu:FMUVariable, fmu:InitialParameter, vsso:MotorMaxPower ;
    fmu:hasFMUVariableName "motor_max_power" ;
    fmu:hasDataType xsd:double ;
    qudt:unit unit:W ;
    fmu:parameterName "motor_max_power" .

req:MotorDefaultRequirementSet a req:ConditionalDefaultRequirement ;
    req:hasID "MOTOR_DEF_COND_001" ;
    req:hasDescription "Default requirements for motors based on power" ;
    req:conditionParameter fmu:MotorMaxPower ;
    req:conditionValue "1500"^^xsd:integer ;
    req:hasRequirementSet req:MotorDefaultRequirementSet .
```

Listing 7: Initial parameter and default requirement set definition.

References

[1] D. Gianni, A. D'Ambrogio, and A. Tolk, Eds., *Modeling and Simulation-Based Systems Engineering Handbook*. CRC Press, 2014. [Online]. Available: https://doi.org/10.1201/b16025

[2] Modelica Association Project, "Functional mock-up interface specification," *Version 3.0.1*, July 2023. [Online]. Available: https://fmi-standard.org/docs/3.0.1/

[3] Modelica Association Project "SSP", *System Structure and Parameterization 1.0.1*, July 2022, version 1.0.1. [Online]. Available: https://ssp-standard.org/downloads

[4] L. Hatledal, R. Skulstad, G. Li, A. Styve, and H. Zhang, "Co-simulation as a fundamental technology for twin ships," *Modeling, Identification and Control*, vol. 41, no. 4, pp. 297–311, 2020.

[5] A. Oprea, R. Hällqvist, L. Knöös Franzén, M. Eek, I. Staack, and H. Gavel, "Connecting system simulation to aircraft concept development," in *Proceedings of the 32nd Congress of the International Council of the Aeronautical Sciences*, Pudong Shangri-La, Shanghai, China, September 2021.

[6] J. Köler, H. Heinkel, P. Mai, J. Krasser, M. Deppe, and M. Nagasawa, "Modelica-association-project 'system structure and parameterization'—early insights," in *Proceedings of the 1st Japanese Modelica Conference*, Tokyo, Japan, May 2016.

[7] R. Hällqvist, R. Munjulury, R. Braun, M. Eek, and P. Krus, "Realizing interoperability between mbse domains in aircraft system development," *Electronics*, vol. 11, no. 2901, 2022.

[8] K. J. Schlager, "Systems engineering-key to modern development," *IRE Transactions on Engineering Management*, vol. EM-3, no. 3, pp. 64–66, July 1956.

[9] INCOSE, *INCOSE Systems Engineering Handbook*. John Wiley & Sons, 2023.

[10] H. Sun, W. Fan, W. Shen, and T. Xiao, "Ontology-based interoperation model of collaborative product development," *Journal of Network and Computer Applications*, vol. 35, no. 1, pp. 132–144, 2012.

[11] S. Mittal and J. L. R. Martín, "Model-driven systems engineering for netcentric system of systems with DEVS unified process," in *2013 Winter Simulations Conference (WSC)*, 2013, pp. 1140–1151.

[12] A. L. Ramos, J. V. Ferreira, and J. Barceló, "Model-based systems engineering: An emerging approach for modern systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 1, pp. 101–111, 2011.

[13] A. Van Lamsweerde, "Requirements engineering in the year 00: A research perspective," in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 5–19.

[14] D. Pandey, U. Suman, and A. K. Ramani, "An effective requirement engineering process model for software development and requirements management," in *2010 International Conference on Advances in Recent Technologies in Communication and Computing*. IEEE, 2010, pp. 287–291.

[15] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, 01 2001.

[16] "Ieee recommended practice for software requirements specifications," *IEEE Std 830-1998*, pp. 1–40, 1998.

[17] S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML: The Systems Modeling Language*, 2nd ed. Morgan Kaufmann, 2012.

[18] J. Cleland-Huang, O. Gotel, and A. Zisman, *Software and Systems Traceability*. London: Springer, 2012, vol. 2, no. 3.

[19] M. Grieves and J. Vickers, "Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems," in *Transdisciplinary perspectives on complex systems*. Springer, Cham, 2017, pp. 85–113.

[20] M. Juarez, V. Botti, and A. Giret, "Digital twins: Review and challenges," *ASME Journal of Computing and Information Science in Engineering*, vol. 21, no. 3, p. 030802, June 2021.

[21] A. Sharma, E. Kosasih, J. Zhang, A. Brintrup, and A. Calinescu, "Digital twins: State of the art theory and practice, challenges, and open research questions," *Journal of Industrial Information Integration*, vol. 30, p. 100383, 2022.

[22] M. Jacoby and T. Usländer, "Digital twin and internet of things—current standards landscape," *Applied Sciences*, vol. 10, no. 18, p. 6519, 2020.

[23] J. Neidig, A. Orzelski, and S. Pollmeier, "Asset administration shell reading guide (as of january 2022)," Plattform Industrie 4.0, Industrial Digital Twin Association, Tech. Rep., January 2022. [Online]. Available: https://www.plattform-i40.de/IP/Navigation/DE/Home/home.html

[24] S. González-Gerpe, A. Cimmino, and S. Bernardos, "An extension of thing descriptions from the web of things for digital twins," *Open Research Europe*, vol. 2, p. 144, 2022, version 1; peer review: 3 approved with reservations.

[25] F. Stadtmann, A. Rasheed, T. Kvamsdal, K. Johannessen, O. San, K. Kölle, and J. O. Skogås, “Digital twins in wind energy: Emerging technologies and industry-informed future directions,” *IEEE Access*, 2023.

[26] M. Vuković, D. Mazzei, S. Chessa, and G. Fantoni, “Digital twins in industrial iot: a survey of the state of the art and of relevant standards,” in *2021 IEEE International Conference on Communications Workshops (ICC Workshops)*, 2021.

[27] S. Cavalieri and S. Gambadoro, “Proposal of mapping digital twins definition language to open platform communications unified architecture,” in *Italian National Conference on Sensors*, 2023.

[28] E. O’Connell, W. O’Brien, M. Bhattacharya, D. Moore, and M. Penica, “Digital twins: Enabling interoperability in smart manufacturing networks,” *Telecom*, 2023.

[29] J. Leng, D. Wang, W. Shen, X. Li, Q. Liu, and X. Chen, “Digital twins-based smart manufacturing system design in industry 4.0: A review,” *Journal of Manufacturing Systems*, 2021.

[30] X. Hu, S. Li, T. Huang, B. Tang, R. Huai, and L. Chen, “How simulation helps autonomous driving: A survey of sim2real, digital twins, and parallel intelligence,” *IEEE Transactions on Intelligent Vehicles*, 2023.

[31] S. J. E. Taylor, C. Macal, A. Matta, M. Rabe, S. M. Sanchez, and G. Shao, “Enhancing digital twins with advances in simulation and artificial intelligence: Opportunities and challenges,” in *2023 Winter Simulation Conference (WSC)*, 2023.

[32] A. M. Madni, C. C. Madni, and S. D. Lucero, “Leveraging digital twin technology in model-based systems engineering,” *Systems*, vol. 7, no. 1, p. 7, 2019.

[33] F. Tao, H. Zhang, A. Liu, and A. Y. C. Nee, “Digital twin in industry: State-of-the-art,” *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2405–2415, 2019.

[34] F. Tao and Q. Qi, “Make more digital twins,” *Nature*, vol. 573, no. 7775, pp. 490–491, 2019.

[35] K. Piirainen, G. Kolfschoten, and S. Lukosch, “The joint struggle of complex engineering: A study of the challenges of collaborative design,” *International Journal of Information Technology & Decision Making*, vol. 11, pp. 1–39, 2012. [Online]. Available: https://doi.org/10.1142/S0219622012400160

[36] J. Köhler, H.-M. Heinkel, P. Mai, J. Krasser, M. Deppe, and M. Nagasawa, “System structure and parameterization–early insights,” *Proceedings of the*

1st Japanese Modelica Conference, pp. 35–42, 2016. [Online]. Available: https://doi.org/10.3384/ecp1612435

[37] C. Gomes, M. Najafi, T. Sommer, M. Blesken, I. Zacharias, O. Kotte, P. R. Mai, K. Schuch, K. Wernersson, C. Bertsch, T. Blochwitz, and A. Junghanns, "The fmi 3.0 standard interface for clocked and scheduled simulations," *Proceedings of the 14th International Modelica Conference*, pp. 27–36, September 2021. [Online]. Available: https://doi.org/10.3384/ecp2118127

[38] F. Yılmaz *et al.*, "Adapting functional mockup units for hla-compliant distributed simulation," p. 98, 2014.

[39] R. Mikelöv and A. Bergsten, "Evaluating the reliability of fmi co-simulation for validation," 2023.

[40] T. Wekerle *et al.*, "Intellectual property protection in smart systems engineering on exchange of simulation models," in *Transdisciplinary Engineering Methods for Social Innovation of Industry 4.0*. IOS Press, 2018, pp. 198–207.

[41] O. Kugu *et al.*, "An fmi-and ssp-based model integration methodology for a digital twin platform of a holistic railway infrastructure system," in *Modelica Conferences*, 2023.

[42] L. Ochel, R. Braun, B. Thiele, A. Asghar, L. Buffoni, M. Eek, P. Fritzson, D. Fritzson, S. Horkeby, R. Hällquist, Kinnander, A. Palanisamy, A. Pop, and M. Sjölund, "Omsimulator: Integrated fmi and tlm-based co-simulation with composite model editing and ssp," *Proceedings of the 13th International Modelica Conference*, pp. 69–78, 2019. [Online]. Available: https://doi.org/10.3384/ecp1915769

[43] G. Sirin, E. Coatanea, B. Yannou, and E. Landel, "Creating a domain ontology to support the numerical models exchange between suppliers and users in a complex system design," in *Proceedings of the ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, 2013*, vol. 2B. American Society of Mechanical Engineers, 2014.

[44] H. Hu, L. Zhang, and C. Ye, "Semantic-based requirements analysis and verification," in *2010 International Conference on Electronics and Information Engineering*, Kyoto, Japan, 2010, pp. V1–241–V1–246.

[45] H. Kaiya and M. Saeki, "Ontology based requirements analysis: lightweight semantic processing approach," in *Fifth International Conference on Quality Software (QSIC'05)*, Melbourne, VIC, Australia, 2005, pp. 223–230.

[46] G. Guizzardi, "Theoretical foundations and engineering tools for building ontologies as reference conceptual models," *Semantic Web*, vol. 1, no. 1,2, pp. 3–10, 2010.

[47] A. Gangemi and V. Presutti, "Ontology design patterns," in *Handbook on Ontologies*, 2009. [Online]. Available: https://doi.org/10.1007/978-3-540-92673-3_10

[48] M. Bravo, L. F. Hoyos Reyes, and J. A. Reyes Ortiz, "Methodology for ontology design and construction," *Contad. Adm*, vol. 64, no. 4, oct./dic. 2019, epub 12-Mayo-2020. [Online]. Available: https://doi.org/10.22201/fca.24488410e.2020.2368

[49] S. Boyce and C. Pahl, "Developing domain ontologies for course content," *Journal of Educational Technology & Society*, vol. 10, no. 3, pp. 275–288, 2007.

[50] M. Achichi, M. Cheatham, Z. Dragisic, J. Euzenat, D. Faria, A. Ferrara, and E. Jiménez-Ruiz, "Results of the 2016 ontology alignment evaluation initiative," in *CEUR Workshop Proceedings*, vol. 1766, 2016, pp. 73–129.

[51] S. Borgo *et al.*, "Dolce: A descriptive ontology for linguistic and cognitive engineering," *Applied Ontology*, vol. 17, no. 1, pp. 45–69, 2022.

[52] H. Pinto and J. Martins, "Ontologies: How can they be built?" *Knowledge and Information Systems*, vol. 6, pp. 441–464, 2004. [Online]. Available: https://doi.org/10.1007/s10115-003-0138-1

[53] J. Brank, M. Grobelnik, and D. Mladenic, "A survey of ontology evaluation techniques," in *Proceedings of the conference on data mining and data warehouses (SiKDD 2005)*, 2005.

[54] A. Gomez-Perez and M. C. Suárez-Figueroa, "Neon methodology for building ontology networks: a scenario-based methodology," 2009.

[55] J. Cao, E. Vakaj, R. K. Soman, and D. M. Hall, "Ontology-based manufacturability analysis automation for industrialized construction," *Automation in Construction*, vol. 139, p. 104277, 2022. [Online]. Available: https://doi.org/10.1016/j.autcon.2022.104277

[56] C. Yang, Y. Zheng, X. Tu, R. Ala-Laurinaho, J. Autiosalo, O. Seppänen, and K. Tammi, "Ontology-based knowledge representation of industrial production workflow," *Advanced Engineering Informatics*, vol. 58, p. 102185, 2023. [Online]. Available: https://doi.org/10.1016/j.aei.2023.102185

[57] J. S. Dong, "Software modeling techniques and the semantic Web," in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 724–725.

[58] R. Chen, C.-H. Chen, Y. Liu, and X. Ye, "Ontology-based requirement verification for complex systems," *Advanced Engineering Informatics*, vol. 46, p. 101148, 2020. [Online]. Available: https://doi.org/10.1016/j.aei.2020.101148

[59] X. Yuan and S. Tripathi, "An approach of dynamically combining ontologies for interactive requirements elicitation," in *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2016, pp. 11–16. [Online]. Available: https://doi.org/10.1109/ICSESS.2016.7883006

[60] S. Törsleff, C. Hildebrandt, M. Daun, J. Brings, and A. Fay, "Developing ontologies for the collaboration of cyber-physical systems: Requirements and solution approach," in *2018 4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC)*, 2018, pp. 25–32. [Online]. Available: https://doi.org/10.1109/EITEC.2018.00009

[61] J. Z. Pan, "Resource description framework," in *Handbook on Ontologies*, R. Studer, S. Grimm, and A. Abecker, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 71–90.

[62] D. L. McGuinness and F. Van Harmelen, "OWL web ontology language overview," W3C, Tech. Rep. W3C Recommendation 10.10, October 2004. [Online]. Available: https://www.w3.org/TR/owl-features/

[63] O. Hartig, C. Bizer, and J.-C. Freytag, "Executing sparql queries over the web of linked data," in *The Semantic Web-ISWC 2009: 8th International Semantic Web Conference, ISWC 2009, Chantilly, VA, USA, October 25-29, 2009. Proceedings*, vol. 8. Springer Berlin Heidelberg, 2009.

[64] G. Prasad and W. A. Milks, "Autonomous verification architectures for complex systems," in *IMETI 2008: International Multi-Conference on Engineering and Technological Innovation, Vol II, Proceedings*, 2008.

[65] S. Sakthivel, "A survey of requirements verification techniques," *J Inf Technol*, vol. 6, pp. 68–79, 1991. [Online]. Available: https://doi.org/10.1057/jit.1991.12

[66] E. Clarke, "Model checking," in *Foundations of Software Technology and Theoretical Computer Science*, ser. Lecture Notes in Computer Science, S. Ramesh and G. Sivakumar, Eds. Springer, Berlin, Heidelberg, 1997, vol. 1346. [Online]. Available: https://doi.org/10.1007/BFb0058022

[67] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *DAC '97: Proceedings of the 34th annual Design Automation Conference*, June 1997, pp. 263–268. [Online]. Available: https://doi.org/10.1145/266021.266090

[68] C. Seybold, S. Meier, and M. Glinz, "Scenario-driven modeling and validation of requirements models," in *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, ser. SCESM '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 83–89. [Online]. Available: https://doi.org/10.1145/1138953.1138969

[69] M. Duzi, M. Košinár, J. Kožusznik, and S. Štolfa, "Knowledge-base approach to software-process development based on TIL," in *Frontiers in Artificial Intelligence and Applications*, vol. 251, 2013, pp. 102–109.

[70] J.-M. Bruel, S. Ebersold, F. Galinier, A. Naumchev, M. Mazzara, and B. Meyer, "The role of formalism in system requirements (full version)," arXiv:1911.02564 [cs.SE], November 2019, submitted on 6 Nov 2019 (v1), last revised 16 Apr 2020 (this version, v6). [Online]. Available: https://doi.org/10.48550/arXiv.1911.02564

[71] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA+ specifications," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, L. Pierre and T. Kropf, Eds., vol. 1703. Berlin, Heidelberg: Springer, 1999. [Online]. Available: https://doi.org/10.1007/3-540-48153-2_6

[72] S. Merz, "The specification language TLA+," in *Logics of Specification Languages*, ser. Monographs in Theoretical Computer Science. An EATCS Series, D. Bjørner and M. C. Henson, Eds. Berlin, Heidelberg: Springer, 2008. [Online]. Available: https://doi.org/10.1007/978-3-540-74107-7_8

[73] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-time Systems*, vol. 2, no. 4, pp. 255–299, 1990.

[74] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[75] C. Bernardeschi, P. Dini, A. Domenici, M. Palmieri, and S. Saponara, "Formal verification and co-simulation in the design of a synchronous motor control algorithm," *Energies*, vol. 13, no. 16, p. 4057, 2020. [Online]. Available: https://doi.org/10.3390/en13164057

[76] T. Grimm, D. Lettnin, and M. Hübner, "A survey on formal verification techniques for safety-critical systems-on-chip," *Electronics*, vol. 7, no. 6, p. 81, 2018. [Online]. Available: https://doi.org/10.3390/electronics7060081

[77] T. A. Pedersen *et al.*, "Towards simulation-based verification of autonomous navigation systems," *Safety Science*, vol. 129, p. 104799, 2020.

[78] F. Meziane, N. Athanasakis, and S. Ananiadou, "Generating natural language specifications from UML class diagrams," *Requirements Engineering*, vol. 13, no. 1, pp. 1–18, 2008.

[79] N. Garanina and O. Borovikova, "Ontological approach to checking event consistency for a set of temporal requirements," in *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*, 2019, pp. 922–927. [Online]. Available: https://doi.org/10.1109/SIBIRCON48586.2019.8958119

[80] M. Musen and P. Team, "The protégé project: A look back and a look forward," *AI Matters*, vol. 1, no. 4, pp. 4–12, June 2015.

[81] M. J. Page, D. Moher, P. M. Bossuyt, I. Boutron, T. C. Hoffmann, C. D. Mulrow *et al.*, "Prisma 2020 explanation and elaboration: updated guidance and exemplars for reporting systematic reviews," *BMJ*, vol. 372, p. n160, 2021.

[82] G. Guizzardi, G. Wagner, J. P. A. Almeida, and R. Guizzardi, "Towards ontological foundations for conceptual modeling: The unified foundational ontology (ufo) story," *Applied Ontology*, vol. 10, no. 3-4, pp. 259–271, 2015.

[83] J. Guerson and J. P. A. Almeida, "Representing dynamic invariants in ontologically well-founded conceptual models," in *20th EMMSAD, CAiSE*, June 2015.

[84] E. Kabaale, L. Wen, Z. Wang, and T. Rout, "An axiom based metamodel for software process formalisation: An ontology approach," in *Software Process Improvement and Capability Determination*, ser. Communications in Computer and Information Science, A. Mas, A. Mesquida, R. O'Connor, T. Rout, and A. Dorling, Eds. Springer, Cham, 2017, vol. 770. [Online]. Available: https://doi.org/10.1007/978-3-319-67383-7_17

[85] G. Sirin, C. J. J. Paredis, B. Yannou, E. Coatanéa, and E. Landel, "A model identity card to support simulation model development process in a collaborative multidisciplinary design environment," *IEEE Systems Journal*, vol. 9, no. 4, pp. 1151–1162, Dec. 2015.

[86] A. Chebieb and Y. Ait-Ameur, "Formal verification of plastic user interfaces exploiting domain ontologies," in *2015 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2015, pp. 79–86.

[87] R. Savolainen, S. Sierla, T. Karhela, T. Miettinen, and V. Vyatkin, "A framework for runtime verification of industrial process control systems," in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 2017, pp. 687–694.

[88] J. Lu, J. Wang, D. Chen, J. Wang, and M. Törngren, "A service-oriented tool-chain for model-based systems engineering of aero-engines," *IEEE Access*, vol. 6, pp. 50 443–50 458, 2018.

[89] B. Bossa, B. Boulbene, S. Dubé, and M. Pantel, "Towards a co-simulation based model assessment process for system architecture," in *Software Technologies: Applications and Foundations*, ser. Lecture Notes in Computer Science, M. Mazzara, I. Ober, and G. Salaün, Eds. Springer, Cham, 2018, vol. 11176. [Online]. Available: https://doi.org/10.1007/978-3-030-04771-9_6

[90] N. Kibret, W. Edmonson, and S. Gebreyohannes, "Category theoretic based formalization of the verifiable design process," in *2019 IEEE International Systems Conference (SysCon)*, 2019, pp. 1–8.

[91] M. Elgharbawy, A. Schwarzhaupt, M. Frey, and F. Gauterin, "Ontology-based adaptive testing for automated driving functions using data mining techniques," *Transportation Research Part F: Traffic Psychology and Behaviour*, vol. 66, pp. 234–251, 2019.

[92] E. Cibrián, R. Mendieta, J. M. Álvarez Rodríguez, and J. Llorens, "Towards the reuse of physical models within the development life-cycle: a case study of simulink models," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–6.

[93] W. Chen *et al.*, "A service-oriented approach supporting model integration in model-based systems engineering," in *Proceedings of the (specific conference name if known)*, April 2023.

[94] A. M. Madni and M. Sievers, "Model-based systems engineering: Motivation, current status, and research opportunities," *Systems Engineering*, vol. 21, no. 3, pp. 172–190, May 2018.

[95] S. P. Kovalyov, *Key Technologies of Digital Twins: A Model-Based Perspective*, 1st ed. CRC Press, 2024, p. 17, first Published 2024.

[96] ——, "Design and development of a power system digital twin: A model-based approach," in *2021 3rd International Conference on Control Systems, Mathematical Modeling, Automation and Energy Efficiency (SUMMA)*, 2021, pp. 843–848.

[97] A. M. Madni, "Mbse testbed for rapid, cost-effective prototyping and evaluation of system modeling approaches," *Applied Sciences*, vol. 11, no. 5, p. 2321, 2021, submitted: 16 January 2021; Revised: 22 February 2021; Accepted: 25 February 2021; Published: 5 March 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/5/2321

[98] J. Lu, G. Tsinarakis, N. Sarantinoudis, G. Arampatzis, X. Zheng, and D. Kiritsis, "A semantic model-based systems engineering approach for assessing the operational performance of metal forming process," *Computers & Industrial Engineering*, 2024.

[99] X. Zheng, X. Hu, J. Lu, R. Arista, J. Lentes, and D. Kiritsis, "An aircraft assembly process formalism and verification method based on semantic modeling and mbse," *Advanced Engineering Informatics*, 2024.

[100] X. Zheng *et al.*, "A semantic-driven approach to support aircraft assembly process formalism and performance analysis," *(to be published in) Computers & Industrial Engineering*, 2024, in Press.

[101] J. Lu, G. Wang, and M. Törngren, "Design ontology in a case study for cosimulation in a model-based systems engineering tool-chain," *IEEE Systems Journal*, vol. 14, no. 1, pp. 1297–1308, March 2020.

[102] E. Siivola, “Requirement verification in simulation-based automation testing,” Master’s thesis, Aalto University, 2016.

[103] P. Bellini, R. Mattolini, and P. Nesi, “Temporal logics for real-time system specification,” *ACM Computing Surveys*, vol. 32, no. 1, pp. 12–42, 2000.

[104] D. Beckett, T. Berners-Lee, E. Prud’hommeaux, and G. Carothers, “Rdf 1.1 turtle: Terse rdf triple language,” W3C, W3C Recommendation, 2014, last accessed: February 25, 2014. [Online]. Available: https://www.w3.org/TR/turtle/

[105] R. Cyganiak, D. Wood, and M. Lanthaler, “Rdf 1.1 concepts and abstract syntax,” W3C, W3C Recommendation, 2014, last accessed: February 25, 2014. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/

[106] L. Yang, K. Cormican, and M. Yu, “Ontology-based systems engineering: A state-of-the-art review,” *Computers in Industry*, vol. 111, pp. 148–171, 2019.

[107] T. Karhela, A. Villberg, and H. Niemistö, “Open ontology-based integration platform for modeling and simulation in engineering,” *International Journal of Modeling, Simulation, and Scientific Computing*, vol. 3, no. 2, pp. 1–36, 2012, accessed: 29.12.2015.

[108] N. Purohit and A. Joshi, “Pellet: A practical owl-dl reasoner,” *Department of Computer Science, Georgia State University, Atlanta*, 2003, accessed: 29.09.2024.

[109] MAN Energy Solutions, *Basic principles of ship propulsion: Optimisation of hull, propeller, and engine interactions for maximum efficiency*, Oct 2018, accessed: 29.09.2024.

[110] A. Ritari, J. Huotari, and K. Tammi, “Marine vessel powertrain design optimization: Multiperiod modeling considering retrofits and alternative fuels,” *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment*, 2023.